University of Leipzig
Faculty of Mathematics and Computer Science
Institute of Computer Science
Department of Natural Language Processing

Scalability of Topic Map Systems
Topic Map Services in a Distributed Environment

Master Thesis

Marcel Hoyer
October 2011

Thesis Supervisors

Prof. Dr. Gerhard Heyer
Institute of Computer Science
University of Leipzig

Dr. Lutz Maicher
Institute of Computer Science
University of Leipzig

Abstract

The purpose of this thesis was to find approaches solving major performance and scalability issues for Topic Maps-related data access and the merging process. Especially regarding the management of multiple, heterogeneous topic maps with different sizes and structures. Hence the scope of the research was mainly focused on the Maiana web application with its underlying MaJorToM and TMQL4J back-end.

In the first instance the actual problems were determined by profiling the application runtime, creating benchmarks and discussing the current architecture of the Maiana stack. By presenting different distribution technologies afterwards the issues around a single-process instance, slow data access and concurrent request handling were investigated to determine possible solutions. Next to technological aspects (i. e. frameworks or applications) this discussion included fundamental reflection of design patterns for distributed environments that indicated requirements for changes in the use of the Topic Maps API and data flow between components. With the development of the JSON Topic Maps Query Result format and simple query-focused interfaces the essential concept for an prototypical implementation was established. To concentrate on scalability for query processing basic principles and benefits of message-oriented middleware were presented. Those were used in combination with previous results to create a distributed Topic Maps query service and to present ideas about optimizing virtual merging of topic maps.

Finally this work gave multiple insights to improve the architecture and performance of Topic Maps-related applications by depicting concrete bottlenecks and providing prototypical implementations that show the feasibility of the approaches. But it also pointed out remaining performance issues in the persisting data layer.

Acknowledgements

I would like to thank Lutz Maicher and the whole Topic Maps Lab team for the discussions and brainstorming sessions around Topic Maps and always being receptive to questions and ideas. Especially, in no particular order: Benjamin Bock, Daniel Seifarth, Michael Prilop, Peter Scholz, Sven Krosse and Uta Schulze. It was (and partially is) a pleasure to work with the Topic Maps Lab team.

Moreover, many thanks to my family and friends for all their patience and helping me to stay the course. Special thanks to Alexander Groß for cross-reading this thesis and Ronja May for inspiring me at recreation time.

Contents

List of Figures

List of Tables

List of Listings

List of Abbreviations

Abbreviation	Description
AJAX	Asynchronous JavaScript and XML
AMQP	Advanced Message Queuing Protocol
API	Application Programming Interface
BSON	Binary JSON
CORBA	Common Object Request Broker Architecture
CPU	Central Processing Unit
CTM	Compact Topic Map Syntax
DBA	Database Administrator
DBMS	Database Management System
DSL	Digital Subscriber Line
DTO	Data Transfer Object
FIFO	First In, First Out
GbE	Gigabit Ethernet
HTTP	Hypertext Transfer Protocol
IPC	Inter-process Communication
ISO	International Organization for Standardization
JAX	Java API for XML
JDBC	Java Database Connectivity
JLI	Java Live Integration
JMS	Java Message Service
JRMP	Java Remote Method Protocol
JSON	JavaScript Object Notation
JTMQR	JTM Query Result
JTM	JSON Topic Map
LAN	Local Area Network
LTM	Linear Topic Map
MOM	Message-Oriented Middleware
MRS	Merge Registry Service
MSMQ	Microsoft Message Queuing
RDF	Resource Description Framework
REST	Representational State Transfer
RMI	Remote Method Invocation

RPC	Remote Procedure Call
RTM	Ruby Topic Maps
RoR	Ruby on Rails
SOAP	Simple Object Access Protocol
SPARQL	SPARQL Protocol and RDF Query Language
SQL	Structured Query Language
TAO	Topics, Associations, and Occurrences
TCP	Transmission Control Protocol
TMAPIX	TMAPI Extensions
TMAPI	Topic Maps API
TMCL	Topic Maps Constraint Language
TMDM	Topic Maps Data Model
TMIP	Topic Map Interaction Protocol
TMQL	Topic Maps Query Language
TMQS	Topic Maps Query Service
TMoR	Topic Maps over RMI
URI	Unified Resource Identifier
VDSL	Very-high-bitrate DSL
WAN	Wide Area Network
WCF	Windows Communication Foundation
WSDL	Web Service Description Language
WS	Web Service
XML	Extensible Markup Language
XTM	XML Topic Maps

Chapter 1

Introduction

1.1 Scope and Environment

1.1.1 Maiana, RTM and MaJorToM

Maiana is a web application created by the Topic Maps Lab to explore and share topic maps. It enables a user to upload topic maps in common file formats (e. g. CTM, XTM, JTM). Uploaded maps can then be browsed by navigating from topic to topic, exploring all attached properties, names and occurrences, and related associations.

The application is an show case and testing platform for many of the technologies the Topic Maps Lab has developed. One essential component of this set is MaJorToM, a Java-based Topic Maps engine that manages accessing the elementary entities of topic maps, so called *constructs*. Additional modules and their functional extension for the MaJorToM core are:

Hatana
: virtual merging of segregated maps

TMQL4J, SesameTM
: querying maps with TMQL or SPARQL

TMAPIX
: importing, exporting of topic maps

JLI
: integration of legacy data stores like Excel or relational databases

TMCL
: validating topic maps against a Topic Maps schema

To demonstrate cross-platform capabilities and integration flexibility, Maiana was implemented on top of the Ruby Topic Maps framework (RTM) using the rapid application development framework Ruby on Rails in combination

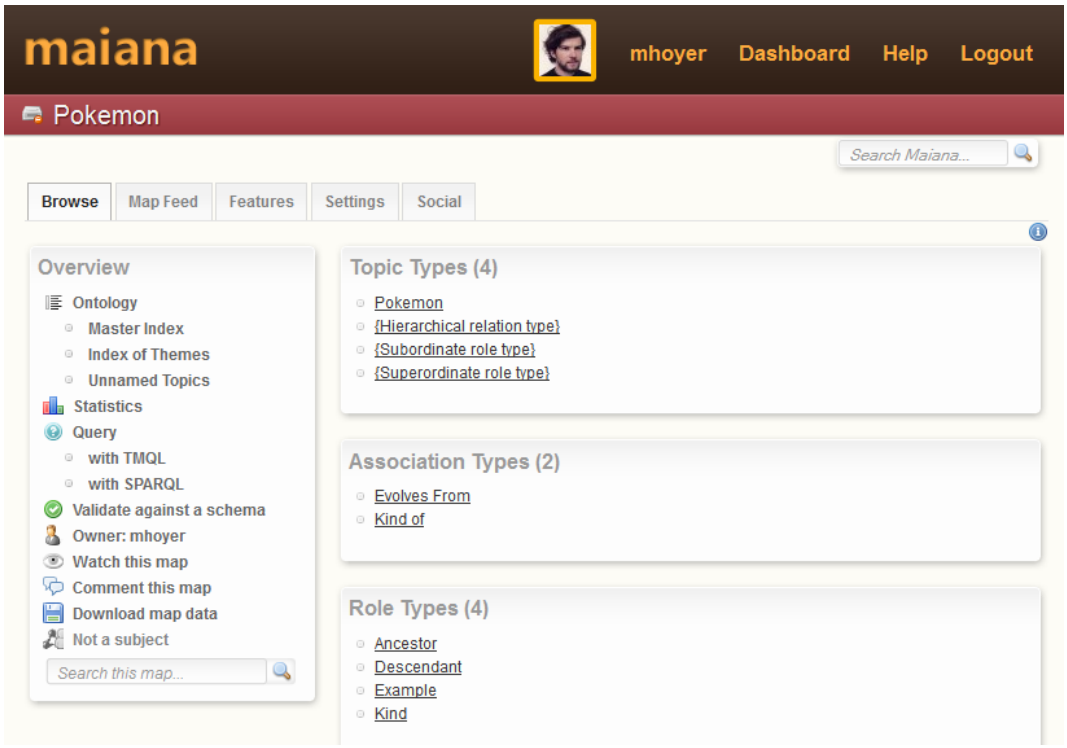

Figure 1.1: Screenshot of Maianas initial Ontology view page.

with some helpful Ruby Gems[1]. This decision in turn required the use of JRuby as the runtime for the web application to enable the integration of the underlying, Java-based components created by the Topic Maps Lab.

1.1.2 Architecture of Maiana

RTM provides a unified accessibility layer wrapping TMAPI-based Topic Maps engines like MaJorToM. In turn, Maiana consumes the functionalities provided by the RTM framework. The system's architecture is depicted in Figure 1.2.

This diagram can be categorized as classic N-tier architecture[2]. Maiana as the Ruby on Rails application represents the front-end tier. RTM implements the business logic layer that in turn consumes the functionalities of the data access layer, played by MaJorToM with in-memory store (or any other TMAPI-based back-end technology). For desktop applications it is common practice to create those kinds of architectures. But looking at large web-oriented, highly scalable services like Twitter or Facebook different application layouts are necessary. The main aspects on those architectures are focused on

[1]Ruby Gems are packages to compose or extend functionality of Ruby application. For instance, Rails is the most popular Ruby Gem.

[2]The detailed architecture of Maiana is more complex. E. g. it uses an additional MySQL database for Ruby on Rails related model data (users, query storage, etc.)

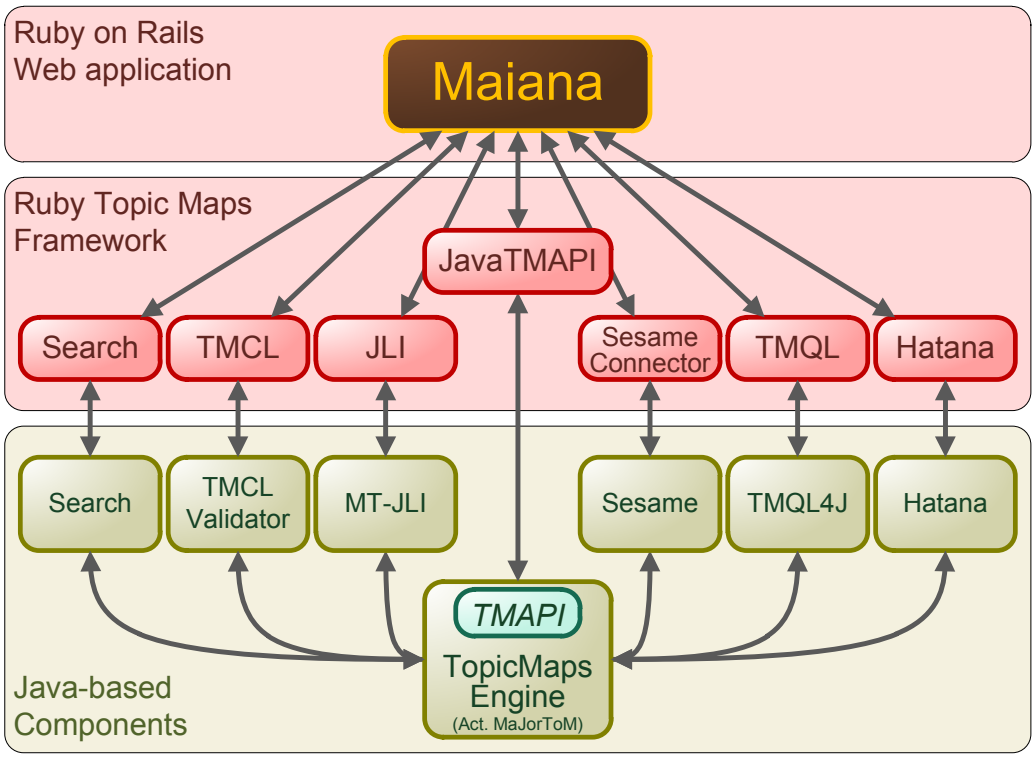

Figure 1.2: Simplified architectural chart of Maiana and the underlying components.

distributing the data and service entry points across a set of machines to reach high throughput even under high load and the ability to manage large sets of data. The possibility to attach additional load-balanced worker nodes[3] to a running system, creates very flexible and extensible services.

1.2 The Problems

1.2.1 Maiana as Single-Process Instance

Due to the straight-forward implementation of the N-tier architecture, the flexibility in scaling data-oriented applications like Maiana is limited. It appears in particular when analyzing the runtime process: even with the encapsulated components and layers of the current architecture, one Maiana instance is started as single, monolithic Java process. Thus it has to manage all objects for the Ruby on Rails front-end, the RTM middle tier and the MaJorToM in-memory back-end. This results in heavy resource consumption with almost no means to solve any performance bottlenecks. Finally, the only way to enhance

[3]Node means computation unit e. g. a process or even a whole server.

the systems performance is to *scale up*. Therefore, it requires more and more hardware resources to achieve reliability.

Besides the resource issues, the monolithic process has other disadvantages too. As of now, it is not possible to replace or even restart separate parts of Maiana when new features are released involving one of the underlying components. This becomes even more apparent, as bugs or not handled exceptions occur in one of the modules. These might crash the whole system. Thus, Maianas online service will become unavailable until a restart is triggered – either automatically or manually.

By now MaJorToM offers a simple solution to separate at least some workload. When using the MaJorToM store implementations for databases (PostgreSQL or Redis) instead of the in-memory store, the data tier can be moved to a different server or a separate process on the same machine. Due to the internals of MaJorToM and the use of TMAPI, other performance issues will arise when fetching data from such persisting store. Those problems will be discussed in Section 1.2.4 and Section 2.2.1.

1.2.2 Concurrent Requests and Stateful Services

Web applications like Maiana must be able to handle several clients simultaneously, even with a growing number of users and a higher probability of concurrent requests. Typically, web servers support those requirements by providing automatic process forking or application pooling. Hence a request should not block the whole web server, otherwise a user might receive a 503 HTTP error.

This behavior can be achieved for Maiana too, by hosting the JRuby-based Ruby on Rails application within an application server like *Glassfish* or *Tomcat*.[4] Hence multiple instances of Maiana are spawned as the number of requests grows. Referring to Section 1.1.2, the current version of Maiana does not use a centralized data storage (like MaJorToM DB or Redis) for the Topic Maps-related data.[5] Because the whole back-end runs in-memory, another architectural problem appears referring to [Fow02]. In fact the application runs as stateful service, holding **all** information in the RAM. This is the root cause for a chain of problems that will be discussed in the upcoming sections.

Furthermore, inspecting the implementation and runtime performance of Maiana, another issue occurs. Rendering a single topic page for complex and large topic maps can take up some seconds due to the time-consuming data collection process in the background. The problem is the result of the syn-

[4]By time of writing Maiana is already hosted inside a Glassfish domain to enhance the reliability.

[5]The performances of current persisting back-end stores for MaJorToM are too slow. Hence the only appropriate configuration for hosting Maiana is the use of the in-memory store.

chronous approach for preparing the content for the views. Using a dynamic AJAX-based front-end could increase the performance perceived by the user, as the basic page layout with initial low-cost information can be provided to the client fast, whereas more expensive to generate content will be received later through dynamic updates of partials of the page. This approach in turn enables the optimization of the servers load balance, as the asynchronism can be used internally for accessing the underlying back-end.

1.2.3 Performance Issues of MaJorToM and Maiana

With the growing list of users and uploaded topic maps, some performance issues occurred from time to time while monitoring the resources of the production instance of Maiana. The underlying MaJorToM was unable or disappointingly slow to manage large topic maps like the `Mondial` with high numbers of topics and associations (TAO $>$ 100000) in memory. Figure 1.3 shows the import durations, CPU loads and memory consumptions for some sample topic maps as result of a profiling session on Maiana.

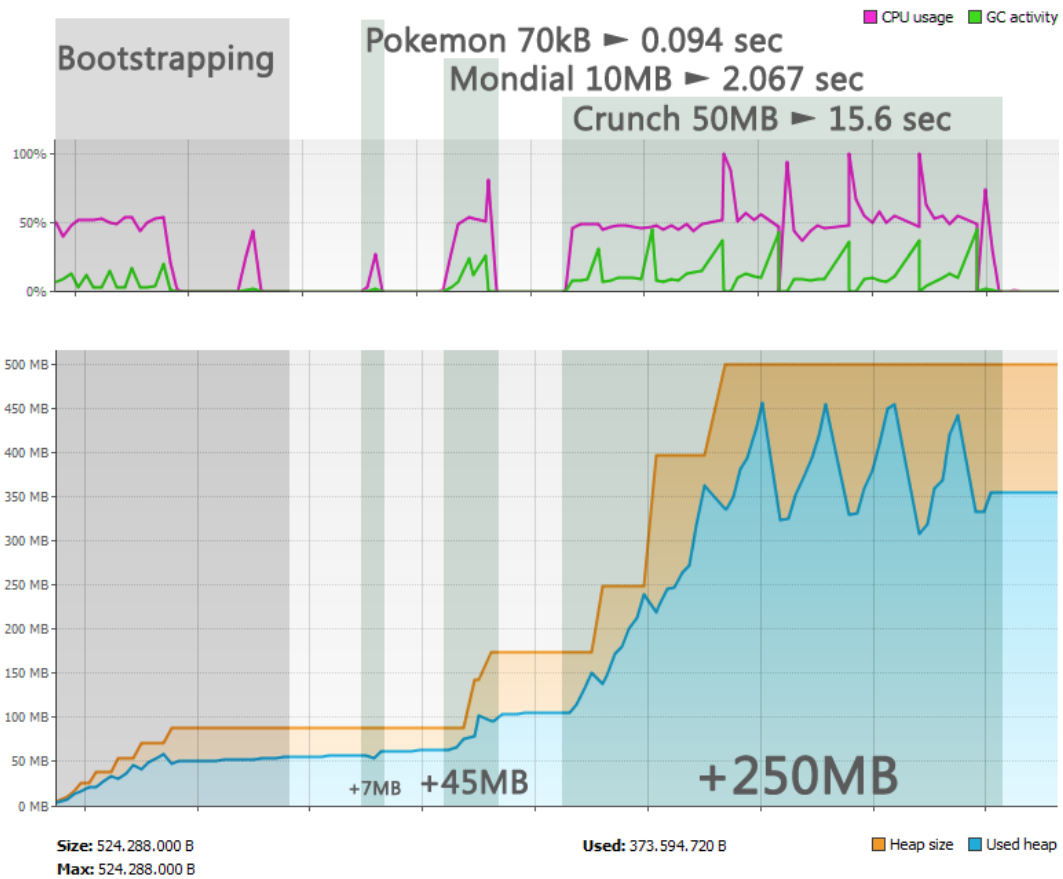

Figure 1.3: CPU usage and memory consumption profile of Maiana

The *Bootstrapping* section on the left side can be ignored as it represents the startup phase of Maiana. After initialization, three sample topic maps were loaded sequentially to measure the memory consumption and CPU load. E. g. loading the `Mondial` topic map from the 10 MB large `mondial.xtm` took ~ 2 seconds and increases the used memory consumption by ~ 45 MB. A detailed list of all measurement values related to this chart can be found in Table 1.1.

Topic Map	∑T	∑A	∑O	∑TAOs	Raw size*	Δ Heap	Duration
Pokemon	198	287	785	1270	36 kB	7 MB	0.15 s
Mondial	5973	16010	9647	31630	540 kB	45 MB	2.1 s
CrunchDB	62242	45568	106310	214120	19.3 MB	250 MB	15.5 s

* Estimated sum over all strings of all locators, names and occurrences.

Table 1.1: Results of profiling the loading of Topic Maps with Maiana. ∑T = Topics, ∑A = Associations, ∑O = Occurrences.

Looking at the stripped-down raw data[6] of i. e. the `Mondial` topic map and comparing its numbers of topics, associations and occurrences (TAO) with the final memory consumption and CPU load shows performance issues. With even larger topic maps (> 100000 TAOs) like the `CrunchBaseDB` topic map, the server requires more and more resources.

Those resource problems can easily be temporarily mitigated by limiting the file size of uploads in Maiana (20 MB as of time of writing), which actually is not a favorable solution. But the problem still exists, as there is no limitation for the overall amount of topic maps. Hence the memory footprint for the increasing number of maps will fill the memory more and more. So even with the given powerful resources of the current production server[7], the web application came to its limits.

Another noticeable performance problem of Maiana is the blocking of the web application, mentioned above. If a topic map is uploaded and opened for the first time, MaJorToM has to convert its contents from the physical file to the in-memory representation. E. g. for a medium-sized topic map, like `Mondial`, loading results in a 2.1 seconds blocking process, during which other HTTP requests to Maiana could not be handled immediately. Thinking of larger maps or higher numbers of users and maps, the user experience of Maiana will be reduced dramatically.

[6] XTM is an XML format to serialize and export topic maps. Hence it needs space for the wrapping markup to describe the actual data (occurrence values, names and locators). E. g. compared to its original 10 MB file size, the extracted pure string values from the `mondial.xtm` only require 540 kB of memory.

[7] Six-core AMD Opteron 2427, 32 GB memory

1.2.4 Provisional, but Inefficient Solutions

The previously mentioned issues might lead to solutions such as concurrent processing of web requests in a multi-threaded or multi-instance environment (e. g. using application servers like Glassfish or Tomcat). But this results in other questions that further complicate matters: How to synchronize the in-memory data across instance boundaries? How to manage concurrent read-/write access on the same data entities?

Even with a threaded or multi-instance solution, the physical resources on servers limit the growth of Maiana, especially when trying to remove the upload limit. Hence scale-up is not the solution. An approach is required to be able to scale-out Topic Maps services horizontally across multiple machines with support for load balancing.

1.3 Solutions in the Remaining Chapters

The following sections will highlight advantages and disadvantages of different technologies that support distributed architectures. With the presentation of prototypical implementations, approaches for solving the scalability issues related to concurrent requests and asynchronous processing will be discussed afterwards.

To limit the scope of this work, concrete performance problems referring Maiana are in focus. Hence the solutions are explicitly concerning with read-only, concurrent access to Topic Maps data, while assuming an environment like Maiana where multiple heterogeneous topic maps need to be managed.

Chapter 2

Possible Approaches

> *"Never underestimate the bandwidth of a station wagon full of tapes hurtling down the highway."* – Andrew S. Tanenbaum.

2.1 Technological Overview

To find solutions for the problems mentioned in Section 1.2 different application architectures and patterns are required. Thinking of common distribution solutions and referring to [Fow02] and [HW03] in general, possible approaches are:

- Distributing database back-ends
- Remote Procedure Calls (RPC)
- Service-oriented architecture (Web Services with SOAP, WSDL)
- RESTful solutions over HTTP
- Message-oriented systems

In [HW03] *File Transfer* is also mentioned to be a solution for sharing information between different nodes. While taking the benchmark results for importing serialized topic maps (see Section 2.2.1) in account, this approach seems not to be applicable for a Maiana like scenario and will therefore not be discussed.

The following sections will provide additional information about upper distribution technologies.

2.2 Distributed Database Back-Ends

Like Ontopia and other Topic Maps engines, MaJorToM can be configured to use databases as the persisting back-end for entire topic maps with all constructs and properties. Thus an easy way for solving the scalability issue

is to apply approved distribution mechanisms of databases e. g. clustering or the master-slave replication principle. Thereby the Topic Maps-providing business layer can be instantiated multiple times, on different nodes, accessing the data tier of a database slave concurrently. Best results should be reached by creating a scenario where each topic map provider is the one and only, exclusively connected client to its own database slave that is in sync with the master database. In combination with an application server as mentioned in Section 1.2.2, Maiana could be instantiated multiple times, where each process has dedicated access to the same datasets across synchronized sources.

Referring to Section 2.3.3 and the following section, Topic Maps engine implementations with database support like MaJorToM-DB still have considerable deficiencies in performance comparing to pure in-memory solutions. This issue is mainly based on the internal use of TMAPI as the upcoming sections will show. [1]

2.2.1 MaJorToM Database Benchmarks

Highlighting the impact for real world scenarios the benchmark results in Table 2.1 show the time consumption for some typical operations in Topic Maps-related environments.

Operation	MT-InMem	MT-Redis	MT-DB*	TinyTiM	Ontopia
▷ CreateTopics	0.87 s	1.42 s	19 min	0.05 s	1.01 s
▷ CreateTopics w/ Name	0.13 s	3.39 s	16 min	0.08 s	0.08 s
▷ CreateTopics w/ N+O	0.23 s	5.59 s	22 min	0.07 s	0.09 s
◁ ReadAllTopics	$\leq$ 0.01 s	0.06 s	0.02 s	$\leq$ 0.01 s	$\leq$ 0.01 s
◁ ReadTopicsByType	0.81 s	0.01 s	0.06 s	$\leq$ 0.01 s	$\leq$ 0.01 s
◁ ReadOccByType	$\leq$ 0.01 s	0.19 s	0.10 s	$\leq$ 0.01 s	$\leq$ 0.01 s
▷ ImportTopicMap**	0.50 s	6.93 s	26 min	0.15 s	0.12 s
◁ ReadImportedTopics	$\leq$ 0.01 s	$\leq$ 0.01 s	0.06 s	$\leq$ 0.01 s	$\leq$ 0.01 s
◁ TMQLTuples	0.06 s	0.10 s	1.44 s	0.01 s	0.01 s
◁ TMQLOccCountFilter	0.32 s	0.42 s	22.41 s	0.29 s	0.27 s

* MaJorToM-DB supports specialized importers bypassing the TMAPI and TMAPIX layer for direct conversion of topic map files into the underlying relational database structure. This speeds up the loading by a factor of 10.

** The same `pokemon.ltm` topic map as in Section 1.2.3

Table 2.1: Durations for different operations in Topic Maps engines

The upper part in Table 2.1 represents the performance for simple TMAPI-based access on Topic Maps engines. The *CreateTopics* operation invokes the `TopicMap.createTopic()` method 1000 times in a loop. No other constructs were

[1] Attempts to run Maiana with a database-enabled Ontopia back-end resulted in bad performance too.

created. Furthermore, the two operations *CreateTopics w/ Name* and *w/ N+O* are creating another 1000 new topics each with an additional single name and accordingly an additional single occurrence. The *ReadAllTopics* test case uses the `TopicMap.getTopics()` method to return the set of all previously generated topics. *ReadTopicsByType* and *ReadOccByType* in turn are addressing the index capabilities of TMAPI.

The lower part of the benchmark result table starts with an empty topic map. It covers results that are more related to scenarios, found in applications like Maiana. The initial import of a serialized topic map using *TMAPIX* is represented by *ImportTopicMaps*. The invocation of *ReadImportedTopics* is equivalent to *ReadAllTopics*, but operates on the previously initialized topic map containing all constructs of the `Pokemon`. The last two TMQL measurements reflect the efficiency of TMQL4J on top of the underlying Topic Maps engines. The following statements were executed against the `Pokemon` topic map.

```
# TMQLTuples
"Pokemon" << atomify << characteristics
  ( . ,
    fn:best-label(.) ,
    fn:best-identifier( . , "true"),
    . >> instances)

# TMQLOccurrenceCountFilter
tm:subject >> instances
  [fn:count( . >> characteristics tm:occurrence ) > 4]
```

Listing 2.1: Sample queries for benchmarking TMQL on Pokemon topic map

The benchmarks show large differences between in-memory solutions and the MaJorToM database approach. The following section explains why.

2.2.2 TMDM-Based Relational Models for Topic Maps Persistence

The main problem of the MaJorToM database back-end is the almost 1:1 mapping of the object-oriented TMDM (Topic Maps Data Model [TMD08]) structure to the underlying relational database schema. Thus the access on Topic Maps constructs to create, read or modify, ends up in short, but very frequent invocation of statements against the database. E. g. by analyzing the log file for a single TMAPI call `TopicMap.createTopic()`, 7 statements (4 `INSERT` and 3 `SELECT`) are issued internally while processing the method. With each statement lasting 2-3 ms, the atomic feature of creating a topic takes around 20 ms. Furthermore, the measured CPU times are just covering the internal database operations and do not include latencies for TCP connections or the

JDBC stack.[2]

Even worse is the situation for the creation of a *typed* topic with `TopicMap .createTopic(type)`. The modified call now produces 21 requests against the database with a nearly linear impact on the duration (about 70 ms in sum).[2] Thinking of more complex processes, like the import of topic maps from disk, explains the outstandingly poor performances for MaJorToM-DB as depicted in Table 2.1. Referring to [Tan02] and the mismatch between latency and throughput, this scenario emphasizes the suboptimal ratio by mapping TMAPI straight forward to SQL statements. In Section 2.3.3 this issue will occur again but in a different environment.

2.2.3 Optimizing Database Back-Ends

Given the database-based setup of Maiana, the first attempt to improve the runtime was to inspect the behavior of the persistent back-end of MaJorToM. Profiling the Maiana process and analyzing the log files of PostgreSQL[3] made it possible to determine the origin of some bottlenecks (see Appendix Appendix A.1 and Appendix A.2). To fix those issues two common DBA techniques were applied:

- Optimizing the slowest queries itself
- Adding indexes to database relations

In addition MaJorToM provides specialized importers for the database store. Bypassing the TMAPIX-based loaders, topic map files are pushed directly to the relational database. Those fixes partially improve the performance of the back-end. But altogether, the speed of MaJorToM-DB still does not fulfill the requirements to provide a pleasant user experience for Maiana.

Another approach was introduced with [KK10]. Kuribara and Kimura provided a method to optimize the retrieval of Topic Maps constructs using their extended Topic Maps database *TOME*. In TOME, entities are stored in the object-oriented database *db4o* to represent the Topic Maps data structure as described in the TMDM. They assume to use the *tolog* query language to extract data from the TOME back-end. The paper is focused on "the search for topic objects referred by a specific association with a particular topic, specified in the query". Compared to the power of remaining *tolog* syntax and/or TMQL, this seems to be an unnecessarily small subset of the possibilities those languages actually have.

[2] See Appendix A.3 for a complete list of triggered SQL statements.

[3] Log analysis was done by enabling the logging feature of PostgreSQL for long duration queries and using the log analyzer tool *pgFouine* and the query analyzer of *PostgreSQL pgAdmin*.

Furthermore, the fixes for MaJorToM and the optimization attempts for TOME targeted a very specific part of a more complex system without regarding the requirement for distribution. Hence the scope for optimization should not be focused on single aspects – especially for real world scenarios as with Maiana. Thus these modifications may only provide improvements for the short term.

2.2.4 Alternatives

Once again referring to Table 2.1 in Section 2.2.1, the column for MaJorToM *Redis* presents a good alternative for a persistent back-end store. The throughput is much higher compared to MaJorToM-DB and acceptable when compared to the in-memory solutions. This results from the simplicity and efficiency of Redis. Its goals are fast and high frequent data access as it is required by today's Internet services like Twitter or Facebook. From a technological perspective this performance is the result of a simple low-level protocol and an in-memory command queuing strategy to encapsulate slow disk IO operations. Another advantage is the possibility to distribute Redis across different machines easily. This enables multiple clients to access the distributed data concurrently. With regard to all these features, MaJorToM combined with the Redis store seems to be a good solution that is available at the time of writing. Hence this setup will be the base for the approaches presented in the upcoming chapters.

Though there might be even better solutions for the back-end question. For instance, the issue described in Section 2.2.2 still is not solved with the Redis approach: the current implementation of the store is still wrapped by the TMAPI layer that induces a higher communication and management overhead as actually needed. For instance, looking at Topic Maps constructs from an entity or domain perspective it might be more effective to transfer the data between database and Topic Maps engine (and obviously the connected Topic Maps clients) as serialized objects representing the whole topic entities with all its child entities, instead of normalizing all constructs into separate tables. An ideal solution should only require a single request to receive the whole entity of a Topic Maps construct.

Thus another, but untested alternative for the back-end choice can be the use of a document-oriented database like *MongoDB*. It allows to persist structured documents, e. g. whole topic maps, using the binary JSON-equivalent *BSON* as exchange format. Compared to Redis it also supports sharding and replication of the persisted documents.

2.3 RPC to Cross Process Boundaries

> *"First Law of Distributed Object Design:*
> *Don't distribute your objects!"* – Martin Fowler [Fow02, p. 89]

With *Remote Procedure Call* or *Remote Invocation* applications are exposing methods to be invoked by remote instances. Thus a communication across process and even system boundaries can be achieved – so called *IPC* (*inter-process communication*). Common implementations can be found in CORBA, .NET Remoting or Java RMI.

As shown in Section 2.2.2, using the TMAPI definitions as contract for distributed approaches seems to be not the best solution. To give a sample proof, an attempt using RPC is shown in Section 2.3.1. This prototype was realized to get first impressions on:

- the feasibility of mapping TMAPI directly into a distributed system,
- the effort required for the implementation,
- the latency overhead compared to in-memory or persistent back-ends

As MaJorToM is implemented with the Java framework a corresponding prototype implementation was built on top of this technology. Java *RMI* is the *Remote Method Invocation* component of the Java framework. It supports an object-oriented way for describing skeletons of the server side and stubs for the client side by defining interfaces (see [WRW96] for more details). This allows a fast and straight-forward implementation of distributed systems. Therefore, this technique was used to realize a spike[4] on exposing TMAPI over TCP: *TMoR* – Topic Maps over RMI. The exchanged data between server and client is serialized and transmitted using the *Java Remote Method Protocol* (*JRMP*).

[4] A prototype implementation in agile development to evaluate the feasibility of an approach.

2.3.1 TMoR – A Prototype Implementation Based on TMAPI Using RMI

To implement a distributed system using RMI, interfaces extending `java.rmi.Remote` needs to be declared following the so-called *Contract First* principle. In this straight-forward solution, all interfaces of the full TMAPI were combined into a single interface. When designing an remote interface in RMI, it is not allowed to specify multiple operations with the same name. But many methods in TMAPI are overloaded (e. g. `org.tmapi.core.Topic.createTopic()`). Hence a redefinition of the *contract* for the TMoR (Topic Maps over RMI) service was necessary where each overloaded method appears disambiguated. The introduction of a segregated interface can be seen as facade for TMAPI and provides a clean and decoupled architecture – even in the scope of prototyping. Therefore, the original TMAPI sources can be kept unchanged and do not require modifications for each interface declaration to extend these with the `java.rmi.Remote` interface.

```
1  public interface ConstructDTO extends Serializable {
2      String getId();
3
4      String getType();
5      void setType(String type);
6
7      String getValue();
8      void setValue(String value);
9
10     String getDatatype();
11     void setDatatype(String value);
12 }
```

Listing 2.2: The contract for `ConstructDTO` data structure.

Due to the fact that only one interface (`TopicMapService`) exists in TMoR, the TMAPI constructs were almost fully flattened and integrated into the contract. Each construct is represented by an ID string as it is provided by the underlying Topic Maps engine with `org.tmapi.core.Construct.getId()`. In turn, all construct property getter and setter methods (e. g. `Construct.getParent`, `Topic.getNames`, etc.) were represented by a corresponding method that takes the construct ID as first argument. Listing 2.3 shows an excerpt of the contract.

```
package de.topicmapslab.tmor.contracts;
import org.tmapi.core.*;
import java.rmi.Remote;

public interface TopicMapService extends Remote {
  // Mapping of Construct
  public ConstructDTO getParentOfConstruct(String constructId);
  public String getTopicMapOfConstruct(String constructId);
  public void remove(String constructId);
  ...

  // Mapping of Topic
  public String[] getSubjectIdentifiers(String topicId);
  public void addSubjectIdentifier(String topicId,
                                   String locator);
  public void removeSubjectIdentifier(String topicId,
                                      String locator);
  public ConstructDTO createName(String topicId,
                                 String typeId,
                                 String value,
                                 String[] scope);
  public ConstructDTO[] getNames(String topicId,
                                 String typeId);
  ...

  // Mapping of TopicMapSystem
  public String getTopicMapByLocator(String topicMapSystemId,
                                     String locator);
  public String[] getTopicMapLocators(String topicMapSystemId);
  public String createTopicMap(String topicMapSystemId,
                               String locator);
  ...
```

Listing 2.3: Excerpt of the `TopicMapService` interface.

To de/serialize the different construct types one lightweight remote data type class `ConstructDTO` was defined. This *Data Transfer Object* only contains the ID, the type of a serialized construct and two optional fields for `org.tmapi.core.Name` and `org.tmapi.core.DatatypeAware` like constructs.

On the client side the whole TMAPI was implemented as facade for TMoR. Hence each TMAPI call from the client application was transparently converted into a corresponding RPC invocation on the `TopicMapService` stub. The server side skeleton in turn calls methods on the implemented `TopicMapService` runtime, that is transforming the requests into TMAPI calls against the underlying server side Topic Maps engine. Therefore, client and server side need to map between the real/virtual TMAPI objects and the flattened `TopicMapService` interface using an implementation of the `Mapper<From, To>` interface. Figure 2.1 shows the schematic relations and the data flow between the client and server side.

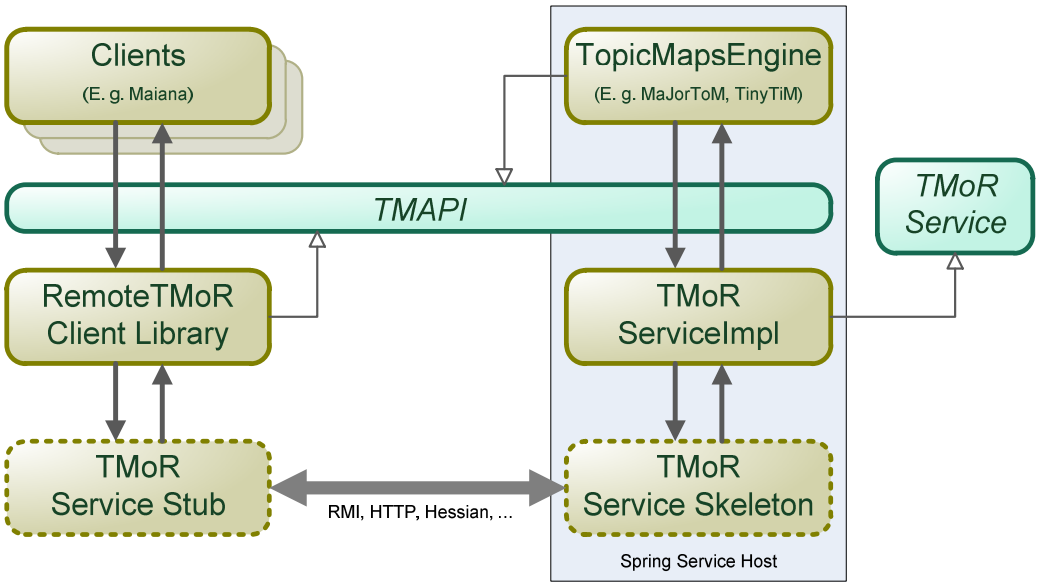

Figure 2.1: Topic Maps over RMI (TMoR) Architecture.

Next to the TMoR interface definition and its straight forward implementation a hosting runtime is required that generates the skeleton and expose the service. Furthermore, the client side stubs need to be generated and must be accessible by the consuming application.

To simplify and decouple the hosting and stubbing process the *Spring Remoting* framework was used. This decision on the other hand allows to easily host the same TMoR service instance over multiple remote technologies: RMI, *HTTP Invoker*, *Hessian* or *Burlap* protocol, *JAX-RPC*, *JAX-WS* and *JMS*. A similar procedure to expose services can be found in the Microsoft Windows Communication Foundation (*WCF*) of the Microsoft .NET Framework.

Due to potential platform or infrastructure issues (like integration of heterogeneous systems or firewall restrictions), the classical communication over RMI that uses the binary *Java Remote Method Protocol* (JRMP) might end up in limited flexibility. Therefore, by choosing the Spring Remoting framework as underlying infrastructure, the publishing of the service with different formats and protocols will be available. Referring the upcoming benchmark results, endpoints for RMI and HTTP Invoker were used to get information about the overhead when using the TMoR service bridge.

2.3.2 Performance Measurement for TMoR

As mentioned in the previous section, the TMoR prototype is prepared for hosting a TMAPI-based service over RMI and/or HTTP using the Spring Remoting framework. To compare the performance of this distributed scenario with the results in Section 2.2.1 the same operations were used. The server

side TMoR service is accessing a MaJorToM in-memory and a MaJorToM Redis instance. To show the relation between the duration and the physical distance of the communicating nodes, the tests were executed with enabled RMI protocol[5] in three different environments[6]:

Local
: Server and client are running on the same machine

LAN
: Server and client are in the same local network (*GbE*)

WAN
: Server and client are connected over a WAN (*VDSL* 25)

	MaJorToM-InMemory			MaJorToM-Redis			
Operation	Local	LAN	WAN	Local	LAN	WAN	Requ.
▷ CreateTopics	2.77 s	4.92 s	53.29 s	3.96 s	6.11 s	51.86 s	2000
▷ CreateTopics w/ Name	2.91 s	5.15 s	80.10 s	5.14 s	7.38 s	82.44 s	2000
▷ CreateTopics w/ N+O	3.86 s	7.08 s	107.29 s	8.54 s	10.43 s	111.87 s	3000
◁ ReadAllTopics	0.64 s	0.56 s	0.56 s	0.33 s	0.16 s	0.15 s	1
◁ ReadTopicsByType	1.36 s	1.51 s	1.64 s	0.10 s	0.26 s	0.37 s	2
◁ ReadOccByType	0.28 s	0.20 s	0.29 s	0.81 s	0.77 s	0.82 s	2
▷ ImportTopicMap*	9.17 s	15.83 s	4.67 min	11.97 s	25.78 s	4.74 min	~ 6400
◁ ReadImportedTopics	0.05 s	0.05 s	0.27 s	0.02 s	0.03 s	0.25 s	1
◁ TMQLTuples	0.15 s	0.23 s	0.44 s	0.15 s	0.26 s	0.46 s	~ 160
◁ TMQLOccCountFilter	0.78 s	1.25 s	6.16 s	1.09 s	1.92 s	6.59 s	~ 320

* The same pokemon.ltm topic map as in Section 1.2.3

Table 2.2: Durations for different operations with TMoR

Comparing the measurements of Table 2.2 with Table 2.1 in Section 2.2.1 the latencies for all operations using the TMoR bridge are obviously visible. But focusing only on the TMoR benchmarks the underlying remote connection has a higher impact on some operation as for others. All modifying operations (▷) like *CreateTopics* or *ImportTopicMap* are lasting increasingly longer the slower the connection[7] gets. In turn the operations to read lists of constructs (◁), transmitting only one or two requests are almost as fast as the in-process benchmark setup in Table 2.1. Those divergences are related to the different number of requests an operation requires in the given test cases. This emphasizes the need for a different distribution-oriented contract for a Topic Maps service.

[5]Referring [Gre08] the performance of the Spring's `HttpInvoker` protocol is similar to RMI. Thus the benchmarks for TMoR are representative for both protocols.

[6]For LAN and WAN the service is running on the same machine as in Section 2.2.1. On the other hand the used client has nearly the same hardware configuration.

[7]Slow connection means smaller bandwidth and a higher distance between the communicating components.

2.3.3 Why TMoR Is Not the Solution

Even though the TMoR solution (with MaJorToM in-memory) is performing better than the MaJorToM Database back-end, it still has architectural deficiencies and performance bottlenecks. In addition the benchmarks for the TMoR prototype were performed in a *segregated environment* where only one client was communicating with one server. How does a single service handle concurrent requests from multiple clients? When instantiating the service multiple times: how should those distributed TMoR instances be managed respecting load balancing, request routing and data synchronization between the worker nodes? Extending the prototypical solution based on Java RMI or the Spring Remoting framework will require heavy implementation steps to fulfill those needs. In conclusion this approach is still not satisfying at all.

Accordingly, the presented way of separating the consuming client process from the data tier only addresses the "one single process" problem referring Section 1.2.1. Looking at the RMI approach from the Maiana perspective the monolithic system can be split into two processes by introducing TMoR. But the management for all topic map resources will then still be bound to a single process – the remote back-end service that seems to be the new bottleneck for concurrent accessibility and resource limitations.

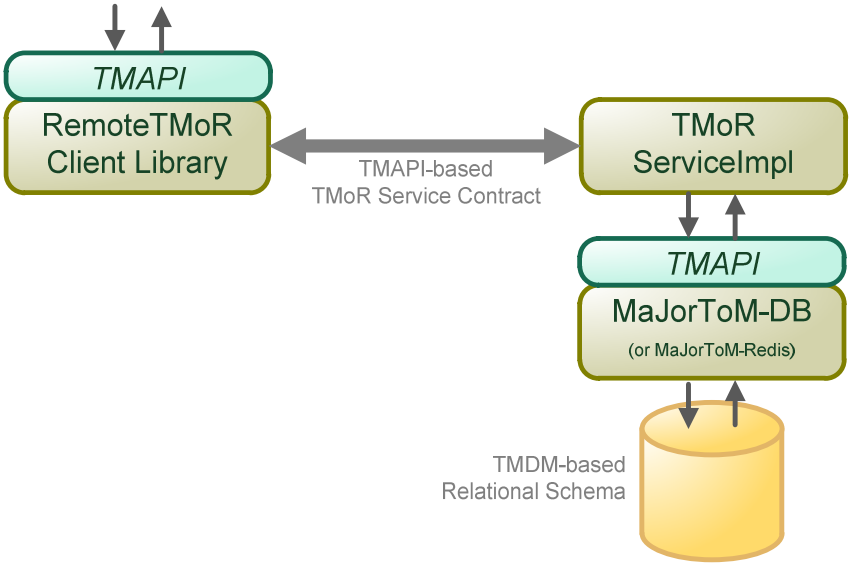

Figure 2.2: TMoR in combination with MaJorToM: misuse of TMAPI or TMDM as intermediate layer for every component.

One might argue to use MaJorToM Redis in combination with TMoR. This setup will then address the issue of concurrent access *and* the resource limitations. But as shown in Figure 2.2 the glue between each layer still is TMAPI or a similar contract (in fact the TMoR Service interface or any other

TMDM-based interface). Referring to "*A fine-grained interface doesn't work well when it's remote.*" [Fow02, p. 89], TMAPI seems not to be the best choice in Topic Maps engines for:

- directly exposing distributed interfaces
- mapping each property call to a database query
- using a TMDM-based relational schema for storing Topic Maps data

Therefore, it seems to be necessary to think about alternative, non TMAPI-centric solutions. Nevertheless, TMAPI has its right to exist and can be implemented by a Topic Maps engine internally to support access to TMDM data structures in an object-oriented style. But distributed Topic Maps solutions need to reduce the number of requests drastically e. g. by using a domain driven approach. Thus serialized DTOs are submitted over the wire once instead of requesting each property as separate remote call. A possible domain-oriented view on the TMDM can be found in Section 2.4.1.

2.4 RESTful and Web Service Approach

In Section 2.3 the prototype TMoR was introduced to show the feasibility and performance of remote Topic Maps systems. With the conclusion that TMAPI seems not to be the right basis for distributing the access on Topic Maps engines, alternatives are required. To embed those new ideas into a more common scenario, nowadays technologies will be used in the upcoming sections. As there are overlapping similarities for possible solutions using the RESTful or web service approach, the following sub sections will cover the basics for both technologies.

2.4.1 A TMDM-Based Data Transfer Structure

In Section 2.3.3 a workaround for more efficient data-oriented communication was mentioned. Trying to submit serialized domain objects instead of asking the remote service for every tiny property will reduce the amount of transmitted requests and therefore eliminate the pricey latencies. The Topic Maps Data Model – TMDM ([TMD08]) is the more or less core specification for Topic Maps-related solutions like the TMAPI, that in turn is the basis for Topic Maps engine implementations. Thus the given data structures of topic maps needs to be analyzed to find a possible solution for a domain-oriented view. Figure 2.3 visualizes the relations between the constructs described in the TMDM. Moreover it tries to inherently show a possible approach on how the constructs might be seen as entities in a domain-oriented perspective.

Starting with the topmost element `TopicMap`, the underlying relations to `Association` and `Topic` are drawn as bold connectors, meaning those child collections are essential parts of a Topic Maps construct and need to be serialized

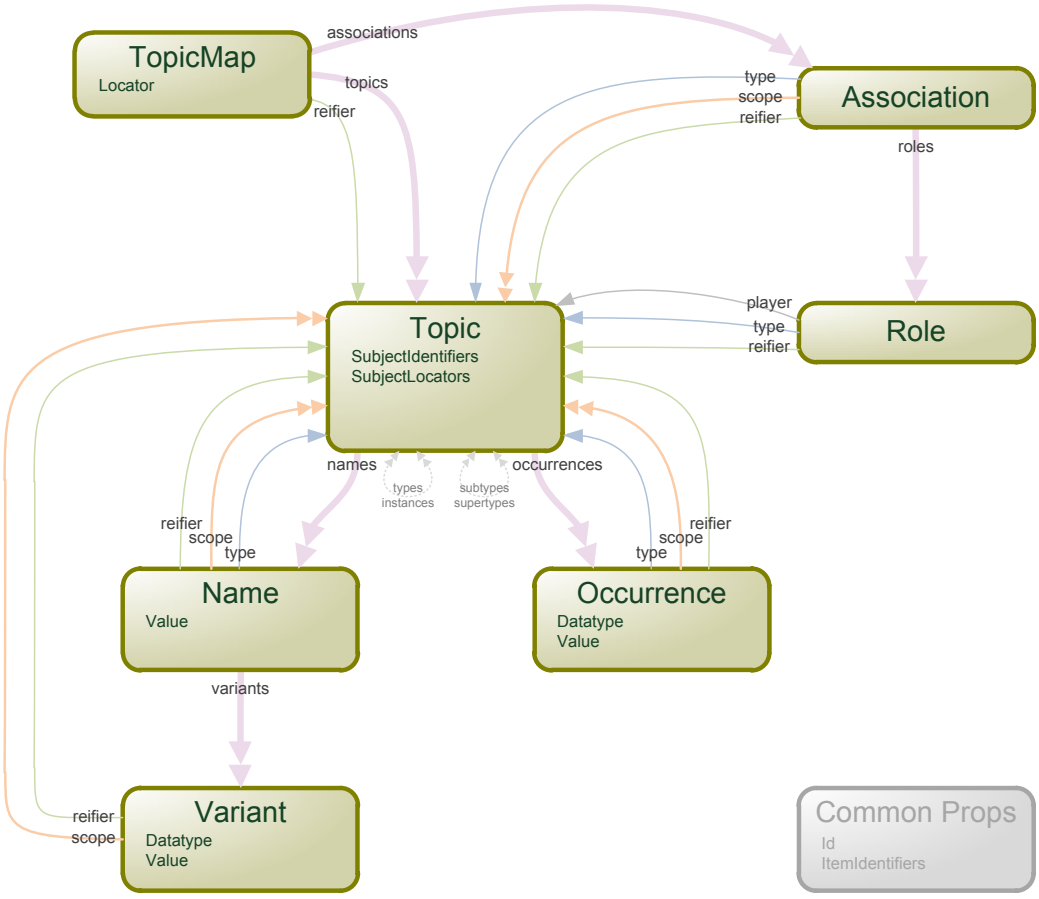

Figure 2.3: Topic Maps Data Model (TMDM) in a nutshell.

when requesting a whole topic map. In turn each `Topic` contains underlying lists of `Occurrence`, `Name` and `Variant`, whereas each `Association` is made of `Role` items. On the other hand all slim arrows in the diagram represent construct references instead of fully serialized objects. Those references can simply be construct IDs or in the context of TMDM an item identifier (or subject identifier/locator in the special case of topics).

Robert Cernys proposal [Cer10] for the exchange format JSON Topic Maps (JTM) almost fulfills the upper idea about serializing Topic Maps constructs. Compared to other formats like XTM or CTM, JTM allows the direct serialization of all types of Topic Maps constructs. Thus the root element must not be `TopicMap`.

All in all, this method of serializing Topic Maps data results in a good bal-

ance between transmitted data size and corresponding request quantity when trying to extract information out of topic maps. For sure this approach might not match within all scenarios. For example, retrieving all fully serialized topics of a huge topic map[8] might lead to a large data stream for a single `GetTopics` request. Especially when thinking in an TMAPI-based manner for accessing Topic Maps data the Listing 2.4 gives a sample where possible problems can occur.

```
1 TopicMap remoteTopicMap = remoteTopicMapSystem.getTopicMap(x);
2
3 for(Topic topic : remoteTopicMap.getTopics())
4 {
5   System.out.println(topic.getId());
6 }
```

Listing 2.4: Retrieving the IDs of all topics.

The main purpose of this sample is to retrieve a list of construct IDs for each topic of a remote topic map. But the invocation of `remoteTopicMap.getTopics()` would to fetch a fully serialized list of all topics, containing all identifiers, names, variants and occurrences. In fact the call of `remoteTopicMapSystem.getTopicMap()` in line 1 already triggers a complete transmission of the whole topic map, when thinking of a thoughtless, straight forward implementation of a RESTful system using the upper serialization idea. Thus the possibilities about serialization of topic map partials is nothing worth if the interface to address just the essential information does not exist. That's where TMQL comes into play. The following section shows how to collect fragments (e. g. just single properties of constructs) of topic maps without moving large datasets between nodes.

2.4.2 TMQL to Request Data

Different data stores like relational databases or XML databases offer query languages to collect specific data from large data sets. The most commonly known one might be *SQL* (or *XPath* for XML-related querying). Those languages allow the creation of statements to request a well defined sub set from the whole data store. This includes filtering, grouping, mapping (projection) and aggregation. Hence it is possible to extract complex, relational parts of all record sets or actually simple aggregated values over multiple rows. The advantage of these query languages is the way the required data is transmitted between the requesting client side and the data provider (i. e. DBMS). It is

[8] E. g. the full War Diary topic map contains 2109 topics, 12033 associations and 43752 occurrences. Serialization to JTM creates a file sized 11 MB. Considering just the serialization of the topics as references should reduce the data size drastically.

possible to define a single statement that – submitted once – will return all requested data in a bulk transmission. Compared to the already questioned object-oriented approach like TMAPI, this reduces the request-response traffic from potentially hundreds to just one single invocation. Thinking of distributed environments the use of a query language should solve the issue of latency between communicating systems.

With the Topic Maps Query Language (TMQL), a query language for Topic Maps data structures is going to fill the gap. The development of the syntax is specified in the ISO 18048 standard, that is part of the Topic Maps standards family. An implementation can be found in TMQL4J that is developed by the Topic Maps Lab and the underlying component required for the TMQL feature of Maiana. Hence TMQL4J will be in focus for upcoming discussions and implementations in this work.

Next to TMQL, other solutions try to support a query-oriented access to Topic Maps engines. Referring to [Gar05] *tolog* is an earlier attempt to query topic maps using a *Prolog* like syntax. An implementation can be found in the Ontopia Topic Maps suite. On the other hand with [Ahm09] an approach to use *SPARQL* as query interface was introduced.

2.4.3 Query Results with JTMQR

In Section 2.4.1 the need and solutions for topic map serialization in distributed environments was discussed. A possible format to export TMDM constructs can be found in the previously mentioned JSON Topic Maps by Robert Cerny ([Cer10]). Under the aspect of TMQL, the support of fragmented serialization of Topic Maps constructs in JTM is an essential feature to enable the creation of response messages of query results. Furthermore, TMQL allows statements not only returning Topic Maps constructs, but single value types[9] and projected lists (tuples). Hence a different or extended format for serializing those query results is needed. In [Bar05] Robert Barta presents an approach for serializing TMQL query results for the *Topic Maps Interaction Protocol* (*TMIP*) using an XML-based data structure. To combine the given features of *XTM Tuple Sequence Encoding* with the compactness of JTM the *JTM Query Results* format (*JTMQR*) was developed for upcoming Topic Maps Query Services. It removes the noisiness of XML and still supports the serialization of tuples, value types, Topic Maps constructs and arrays.

The Listing 2.5 shows an excerpt of an JSON document reflecting a query result set for a given TMQL query sample. The initial version 1.0 of JTMQR was defined using explicit hash keys (`"s"`, `"i"`, `"n"`, etc.) as a more or less direct mapping of the XTM Tuple Sequence Encoding elements to JSON. But as JSON inherently supports data typing and due to the noisiness of the

[9] Compared to the complex data types (any Topic Maps construct) described in the TMDM, value types are simple data types like String, Integer or Float.

```
// possible result for TMQL query:
//   tm:subject >> instances (
//       . ,
//       . >> characteristics tm:name,
//       fn:count( . >> characteristics ) )

{ "version":"1.0",
  "metadata": {"columns":3, "rows":20, "aliases":[]},
  "seq":[
    {"t":[
      {"i": // each topic is serialized with JTM
        { "version":"1.1",
          "prefixes":{"tmdm":"http://psi.top.../model/"},
          "item_type":"topic",
          "subject_identifiers":[ .. ],
          "instance_of":[ .. ],
          "names":[ .. ]}},
      {"s":"Foo"},
      {"n":1}
    ]},
    {"t":[
      {"i": /*JTM*/ },
      {"s":"Zoo"},
      {"n":2}
    ]},
    ...
  ],
  "ordered":true }
```

Listing 2.5: Sample of a serialized JTMQR 1.0 document.

first version 1.0, a stripped-down, cleaner version was released afterwards as JTMQR 2.0.[10] Listing 2.6 shows the corresponding JSON document with the new syntax. Appendix A.4 provides the JSON schema definition for JTMQR 2.0.

With the introducing `metadata` field some information about the generated rows and columns are provided to the consumer. Furthermore, the current TMQL4J implementation has an TMQL extension to enable the definition of column headers within the query, that in turn will be serialized to the `aliases` field. The `tuples` field wraps the index ordered array of tuples, where each entry contains all values (columns) for a single result row. Each value in a tuple can be of any JSON value type: string, number or boolean – or a fully serialized JTM fragment.

[10] See http://code.google.com/p/tmql/wiki/JTMQR to find the full specification with a concrete list of changes.

```
// possible result for TMQL query:
//   tm:subject >> instances (
//       . ,
//       . >> characteristics tm:name,
//       fn:count( . >> characteristics ) )

{ "version":"2.0",
  "metadata": {"columns":3, "rows":20, "aliases":[]},
  "tuples":[
    [ {"jtm": { // each topic is serialized with JTM
        "version":"1.1",
        "prefixes":{"tmdm":"http://psi.top.../model/"},
        "item_type":"topic",
        "subject_identifiers":[ .. ],
        "instance_of":[ .. ],
        "names":[ .. ]}},
      "Foo",
      1
    ],
    [ {"jtm": /*JTM*/ },
      "Zoo",
      2
    ],
    ...
  ],
  "ordered":true }
```

Listing 2.6: Sample of a serialized JTMQR 2.0 document.

TMQL4J 3.1.0 already supports JTMQR by introducing the `USE JTMQR` statement. To get the results of a query returned as plain JSON serialized string, the TMQL statement only requires an additional `USE JTMQR` statement as shown in Listing 2.7.

```
1 tm:subject >> instances
2 USE JTMQR
```

Listing 2.7: Sample to show the use of USE JTMQR statement.

Using TMQL to specify the request and JTMQR (or the XTM Tuple Sequence Encoding) as format for the response seems to be the best solution for the required needs. It allows precise extraction of information out of a topic map and a compact, fast way of result retrieval. Referring Listing 2.4 the task of getting a list of construct IDs for all topics of a topic map can now be achieved with an equivalent TMQL request as in Listing 2.8. The benefit on applying this approach is the concrete retrieval of the distinct data without transferring unnecessary overhead of fully serialized construct structures.

```
1 tm:subject >> instances ( . >> id )
```

Listing 2.8: Retrieving the IDs of all topics using TMQL.

Next to a pure TMQL-based solution with a single `query` operation, additional extensions of the remote service are possible. For instance, providing a RESTful interface to directly address concrete entity resources in a topic map (e. g. `GetOccurrenceValuesOfTopic(‘‘si:http://test’’)`). But this approach might create performance issues, if the developer of a data consuming client application uses the service in a wrong purpose. Exposing almost every resource of a topic map in a TMAPI fashioned style can lead to the problems already discussed in Section 2.3.3. Hence it is the developers job to correctly utilize the service. Otherwise, the client application will not perform well or – even worse – the server application can slow down, not providing the data as fast as it should. But this problem of misuse can also apply to a query-only TMQL approach without the RESTful extensions.

2.4.4 Web Service

As already mentioned in the introduction of Section 2.4, the upper paragraphs already discussed the underlying basis for a possible service interface to provide access to topic maps stored in a distributed environment. On the level of interface definition and by ignoring the specific protocol definitions, a RESTful service can be seen as a specialized Web Service with predefined operations (`GET`,

`POST`, `PUT`, `DELETE`) and a given resource addressing schema. From a contract-first perspective well defined resource-oriented interfaces can be used for web and RESTful services in the same manner. It actually only requires a mapping description between REST (HTTP verbs and resource location paths) and the corresponding service operations – similar to the routing specifications in Ruby on Rails. Hence there seems to be only little effort required to publish a RESTful service as a SOAP-oriented version and vice versa. Using frameworks like Spring Remoting framework or the Microsoft .NET Windows Communication Foundation, it might require just some configuration to expose a single service implementation for different technologies.

In addition RESTful services are very lightweight for different aspects compared to SOAP. The communication protocol does not include such heavy headers and wrapping XML structures. Furthermore, the fixed set of service operations (`GET`, `POST`, `PUT`, `DELETE`) result in fast, optimized RESTful services components on top of back-end systems. Within the scope that topic maps *are* plain and well structured resources, the RESTful approach seems to be the most efficient solution compared to the web service-oriented approach. But the possibilities to host the same service implementation with multiple technologies improves the integration capabilities over different heterogeneous systems.

2.4.5 Summary

The basic idea in this section, to use TMQL as request and JTMQR as response message type of a `query` operation, seems to be a balanced alternative for creating the core access to remote systems. It enables a specific and dedicated way to extract concrete data from topic maps. Thus it can also be applied to the RMI solution in Section 2.3 to reduce the amount of request when implementing a TMDM-like data access.[11]

But another issue to provide a high available and scalable system was not touched: the need for concurrent and asynchronous requesting of Topic Maps data. The previously mentioned technologies like RESTful/Web service or RPC do not automatically cover these requirements as the underlying architecture is built on a request-response model. Therefore, the following chapter Chapter 3 introduces a new architectural solution that addresses these needs in its core.

[11] Moreover, TMQL might include the possibility for distributed query handling by using a *MapReduce*-like approach by dissecting the query tree into sub queries. See Section 4.3.

Chapter 3

Message-Oriented Querying Service

In Chapter 2 the needs and an attempt to break the TMAPI-braced view on Topic Maps engines were shown. Thus TMQL was nominated to become the core operation for a Topic Maps service interface using serialization formats like JTM and JTMQR for the query and data exchange. The chapter also introduced communication technologies that may expose the services around TMQL. Hence a RESTful service like TMIP can already be used to accomplish the call for efficient remote systems.

But this thesis is focusing on scalability features for distributed systems. With Maiana as Topic Maps data consuming web application, a high quantity of requests might hit a future TMQL service. As queries against Topic Maps stores can last between milliseconds and minutes[1], neither the requested service nor the client application should block. This leads to the requirement for asynchronous access to Topic Maps information, that in turn should provide concurrent querying. As Maianas core purpose is to host many topic maps of different sizes in a Social Web like way the underlying back-end needs to distribute the data, but should provide a fast, balanced access layer.

These claims can already be achieved by using the presented technologies and approaches from Chapter 2. With more or less effort a RESTful or simple TMQL service can provide powerful access to topic maps. But referring Maianas requirements, a message-oriented approach might fit better into this scenario. Section 3.1.1 will show pros and cons for such solution.

[1] Next to the pure processing time for a TMQL query additional resources are needed. For instance, the creation and the exchange of the result message takes time. See "Evaluation" section in [Bar05].

3.1 Technological Overview

3.1.1 Message-Oriented Middleware

Given by the name, *message-oriented middleware* (*MOM*) is all about systems that are used to let participants communicate in distributed environments using messages. These systems include dedicated soft- and/or hardware components to handle the transmission of messages. In addition MOM is based on message centric architectures and patterns that constitute the basics for clean interfaces and the ability to create cross-platform applications with less effort.

The main purpose of such middleware component is to provide reliable and high scalable services to allow communication between multiple partners in a heterogeneous, remote environment. In addition features for routing, persistence, transaction, failure handling and transformation are parts of message-oriented middleware.

Compared to other distribution technologies like REST or Web Services, MOM has advantages, when it comes to scalability requirements. It is highly optimized for a high message throughput and inherently supports load balancing. Depending on standards and concrete MOM implementations, the message-oriented approach also supports the following features:

- simple interoperability between participants by providing an simple API
- cross-platform integration of different applications
- flexible extensibility of runtime systems as additional worker nodes can be attached ad hoc
- persistence of messages
- support for transactional processing
- secure communication with encryption, authentication and authorization

Before applying an additional component to a software architecture the deficiencies also has to be checked. A message-oriented solution needs a dedicated middleware in the runtime environment that requires administration (configuration and maintenance) and possibly can produce failures affecting the whole system. Without question, transmitting messages via MOM also creates additional latencies between the communicating participants and needs further resources like memory and processor use. But the advantages for scalability possibilities constitute a solution using an extra component.

To develop message-oriented solutions some standards already exist, which specify a set of features and an API. The common ones are the *Advanced Message Queuing Protocol* (*AMQP*) and *Java Message Service* (*JMS*). Due to a more generic definition of an message-oriented architecture and therefore better interoperability support, AMQP was chosen for the research in this thesis.

3.1.2 Available Solutions

A number of products exist to create message-oriented applications. The well-established ones are Apache ActiveMQ/Qpid, IBM Websphere MQ, Microsoft MSMQ or RabbitMQ. Each of those support multiple features and provide different APIs to access the messaging functionalities.

In the scope of this thesis the MOM to be chosen should be free (preferably open source), well-documented and easy to install and maintain. As Maiana is a Ruby on Rails application using Java-based back-ends another aspect for selecting the right product was the support of client libraries for Java and Ruby. A first attempt using ActiveMQ did not satisfy referring the specified needs – especially the given Ruby client. Instead the more recent competitor RabbitMQ fulfills the requirements and has some further advantages:

- Leading implementation of AMQP specification.
- Easy installation process compared to others: download, extract and run. No initial configuration required for a default setup.
- Support of many client libraries in different programming languages.
- Written in Erlang – a functional programming language with focus on concurrency and message passing as communication paradigm between processes.
- Different benchmarks accomplished a throughput of 10k messages per second.
- Provides an additional simple web front-end for queue management and monitoring.

But the nomination of RabbitMQ should not be seen as final decision. For this thesis the development of a prototype with the need for a message-oriented middleware is in focus. Hence the available products should be evaluated in detail if a more concrete implementation of a distributed Topic Maps system will be the result of this work.

3.1.3 Principles of Message-Oriented Middleware

Different standards and implementations of message-oriented middleware provide specific feature lists and naming conventions. The following explanations are focused on the namings in the scope of RabbitMQ and AMQP.

Message Broker

The central component of a MOM solution is the so-called *message broker*. It manages the infrastructure and transmission of the messages. In distributed environments multiple connected message brokers can be instantiated to enhance reliability and scalability.

Producers and Consumers

The participating clients in a message-oriented system (connected to the message broker) are the *producers* and *consumers*. While consumers are listing on queues for incoming messages, a producer is creating and sending messages to an exchange point. Consumers will be notified about new, incoming messages by the MOM. From a conceptual perspective, a client can be a consumer and producer in one instance. This allows to reproduce a request-response behavior as it is commonly known for web service operations. But the basic idea for message-oriented architectures is to create decoupled, independent components that either provide or react on messages. This concept is a key element to create concurrent, asynchronous and scalable services.

Messages

Messages are the wrapping hull of data that is send between the participating parties in a remote environment. In the scope of AMQP and obviously most other standards or protocols like JMS, SOAP or HTTP, a message consists of a header with additional information as key-value pairs and a the actual payload.

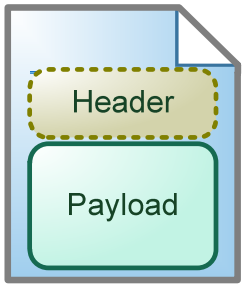

Figure 3.1: An AMQP message with header and payload.

The payload in AMQP is a simple byte stream and must not match any concrete data structure. Hence the message format between consumer and producer can be defined by the individual needs. In general a plain, platform-related, binary serialization of objects, simple string messages or the use of today's formats like JSON, BSON or XML are possible ways to exchange data.

Headers

The headers of messages can be compared to those of the HTTP or SOAP protocol. They contain properties, like the routing key, priority, delivery type or expiration of the message, that is relevant for the message broker itself.

Further meta information that is relevant for the consumer can also be specified in this section. This includes i. e. a message correlation id, reply information or custom meta data.

Queues

Each message sent by a producer won't be forwarded directly to its recipient, but stored in a *queue*. Those reflect the so called *FIFO* (*First In, First Out*) principle, where incoming messages are collected and released (received by the consumer) keeping the chronological order. This is the basis for an easy asynchronous, load balanced exchange of messages, as multiple consumers can listen to the same queue while sharing the workload in a *round-robin* fashioned way.

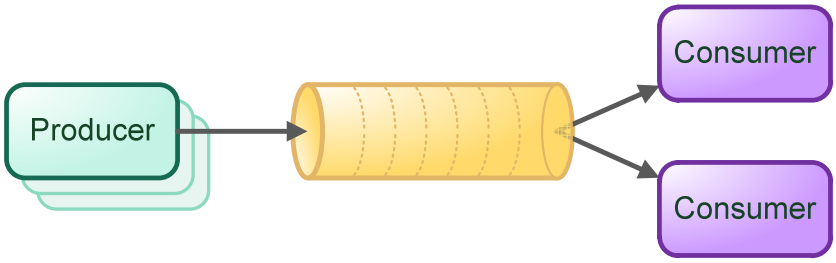

Figure 3.2: A worker queue with two consumers.

In addition AMQP specifies different properties for a queue to enable features like durability, exclusive use or auto-delete if no more consumers are connected. The message broker manages the life cycle of the queues to assure their functionality.

Exchanges, Bindings and Routing

RabbitMQ encapsulates the queues with further abstraction elements – the so-called *exchanges*. Instead of sending messages directly to queues the producer submits to an exchange that works like a delivery service point, where the message broker is dispatching the message header and forwarding the message to its addressed destination queues. The configuration for this routing is defined with a so called *binding* between a queue and an exchange. Thus the header of an incoming message has to match the routing key and the queue name to be forwarded correctly.

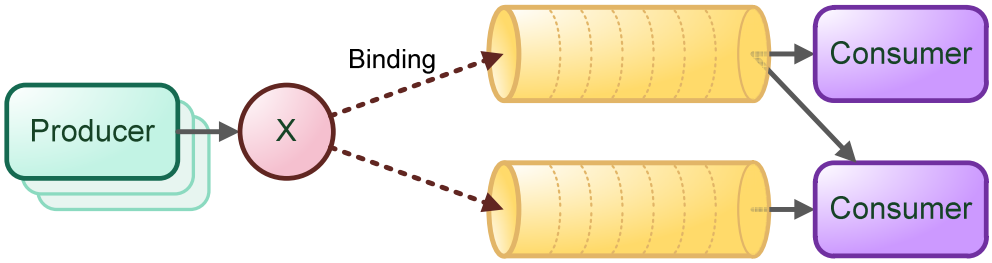

Figure 3.3: An exchange bound to two queues.

Depending on the routing key configuration, four different kinds of exchanges can be created.

Fanout exchange
: The routing key will be ignored for this kind of exchange. Thus all connected queues will receive the message.

Direct exchange
: If the producer submits a message with routing key `foo` only exchanges exactly matching `foo` will be supplied.

Topic exchange
: Works similar to the direct exchange, but allows consumers to listen to routing keys using wildcards. E. g. a consumer listening to `foo.*` will receive message sent to `foo.bar` and `foo.zoo`.

Headers exchange
: The most flexible exchange type that enables routing by comparing possibly all (not only the routing key) header values with a predefined rule set.

```
require 'amqp'
AMQP.start(:host => 'localhost') do
  # create channel, exchange, queue and bind
  channel = AMQP::Channel.new
  exchange = channel.fanout('X-name')
  queue = channel.queue('Q-name').bind(exchange)

  queue.subscribe do |header, msg|    # consume
    p header, msg
  end

  exchange.publish('Hello World')     # produce
end
```

Listing 3.1: Sample of an AMQP consumer and producer.

From a developers perspective, connecting to a message broker like RabbitMQ that implements the AMQP standard can be achieved easily. Different

libraries for almost every platform and programming language are available. E. g. Listing 3.1 shows a sample implementation on how to create a consumer and producer addressing the same queue and exchange. The sample is based on the *AMQP gem*[2] for Ruby. The upcoming implementation of the Topic Maps Query Service will use the Java libraries shipped with RabbitMQ.

Creation and management of queues, exchanges and bindings can also be done using the web front-end of RabbitMQ.

3.2 Implementation Details

Referring Section 3.1 the flexibility in defining message flows and the included load balancing feature of message-oriented middleware are the base for the development of a possible solution for a TMQL-based Topic Maps service. Figure 3.4 shows the architectural overview for the prototype implementation of a Topic Maps Query Service (TMQS) that will be described in detail in the following sections.

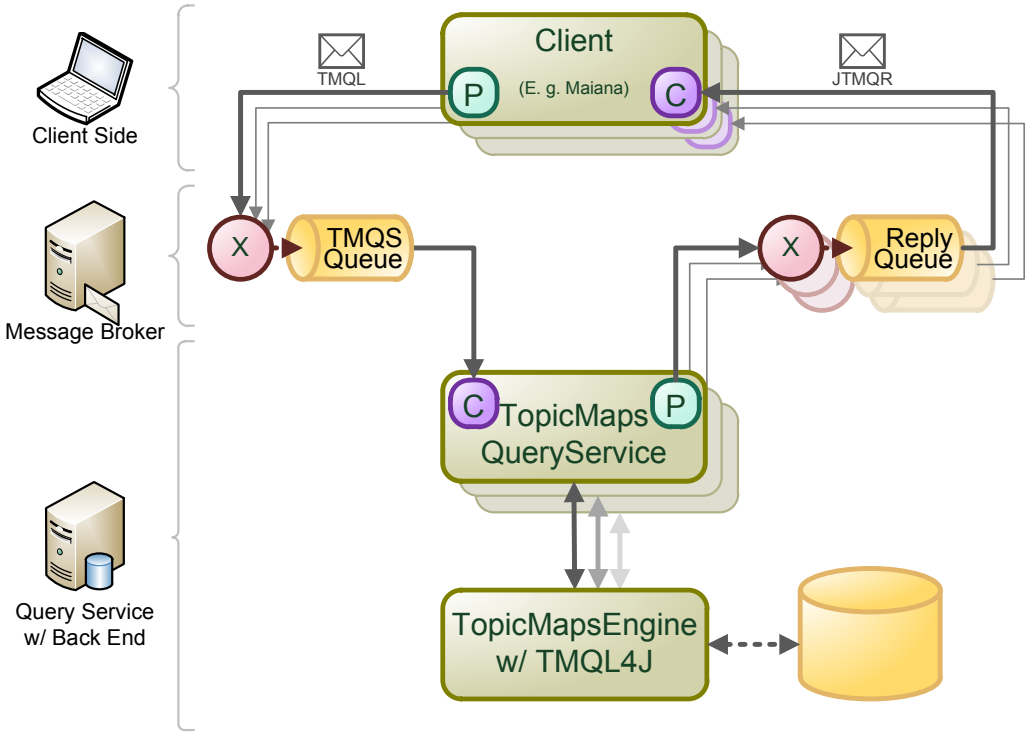

Figure 3.4: Message-oriented approach for the Topic Maps Query Service.

The main purpose of this prototype is to proof the feasibility of a message-oriented solution to achieve:

[2]See https://github.com/ruby-amqp/amqp

Scalability
: by starting multiple TMQS worker processes in parallel as needed.

Extensibility
: by providing flat, easy-to-implement contracts that allow different implementations against the same interfaces (e. g. to provide a simple caching layer or an autogenous routing component).

Flexibility
: due to the features of the underlying message broker. For instance, the routing capabilities may be used to realize the sharding or fragmentation of the Topic Maps stores for multiple TMQL service instances.

The three main layers in Figure 3.4 build up a possible scenario for a Topic Maps-data-consuming application. The upper tier represents the consumer side where multiple clients (e. g. multiple Maiana instances, TMQL Console, etc.) can connect to the centered message broker (i. e. RabbitMQ). To query the underlying Topic Maps Query Service a corresponding message addressing a related TMQS queue has to be published to a message brokers exchange. For simplicity the further explanations are based on a single fanout queue for the query service (but multiple reply queues for the connected clients).

In turn, the bottom tier represents the actual service with its underlying Topic Maps engine that provides the data. By now each service is hosted in a separate console application.[3] Hence it allows a simple scaling of the whole service by starting multiple processes on either one single machine or distributed over multiple nodes. This flexibility allows fast reaction to achieve optimal load balancing to corresponding query traffic.

3.2.1 The Service Contracts

The prototype implementation for this approach combines the achievements of the previous sections. Thus the starting point for the development was a lightweight contract, defining the service interface. It describes a single operation named `query` requiring a `QueryMessage` object as single parameter. This message type encapsulates the actual `query` and the targeted topic map using the `topicMapLocator` field. It is good practice to extract operation arguments and the return value into a separate data structure to provide better extensibility of the service interface and its data definitions. On the other hand the method returns an instance of `IResultSet` as direct result of the processed query.[4]

This contract does not constitute the message-oriented approach. It instead represents the core element for concrete implementations that focus on specific

[3] Hosting the Topic Maps Query Service as background service or inside an application server can be an option for later productive environments.

[4] `IResultSet` is part of *TMQL4J* – the TMQL engine implementation from the Topic Maps Lab.

```
public interface TopicMapQueryService {
    IResultSet query(QueryMessage queryMessage);
}

public final class QueryMessage {
    public String topicMapLocator;
    public String value;
}
```

Listing 3.2: Contracts describing the Topic Maps Query Service and the structure of the incoming query message.

TMQS-related features. For instance, a query caching layer or specialized TMQL runners which are optimized for specific domains or distinctive types of topic maps (e. g. very large vs. tiny, strongly vs. loosely linked maps). Due to the encapsulation by using a contract, a concrete implementation of `TopicMapQueryService` can furthermore be used for other technologies as presented earlier. Hence a RESTful, Web service or RMI-based solution can instantiate the upcoming `TopicMapQueryServiceImpl` component and expose its operations.

3.2.2 The Service Implementation

With the simplicity of the contract, the implementation of the Topic Maps Query Service is straight forward as Listing 3.3 shows. Strictly speaking it is just a composing implementation, connecting an already instantiated TMAPI-based implementation of a Topic Maps engine with the existing `TMQL4J` to get the results for the requested query when the `query` operation is invoked.

```
public class TopicMapQueryServiceImpl
      implements TopicMapQueryService {

  private TMQLRuntimeFactory tmqlRuntimeFactory;
  private TopicMapSystem tms;

  public TopicMapQueryServiceImpl(TopicMapSystem tms) {
    this.tms = tms;
    tmqlRuntimeFactory = TMQLRuntimeFactory.newFactory();
  }

  public IResultSet query(QueryMessage msg) {
    String mapLocator = msg.topicMapLocator;
    TopicMap map = tms.getTopicMap(mapLocator);
    ITMQLRuntime runtime = tmqlRuntimeFactory.newRuntime();

    return runtime.run(map, msg.value).getResults();
  }
}
```

Listing 3.3: Implementation of the Topic Maps Query Service.

The more complex details are addressing the hosting component of the Topic Maps Query Service. To connect to the RabbitMQ message broker, the provided Java client libraries were used. But it requires some wiring between the participating components (Topic Maps Query Service, TMQL4J, TMAPI-based Topic Maps engine and the RabbitMQ client API). Furthermore, the extensibility of the service had be considered when designing the application.[5] Hence gaining a loose coupling, one essential design pattern for this solution was the *Dependency Inversion Principle*. This in turn allows the use of an *Inversion of Control* container (*IoC*) to automatically instantiate all required components for the application context to finally host the Topic Maps Query Service. As the underlying development platform is Java a common implementation of an IoC container can be found in the *Spring Framework*. The following figure shows the dependencies between all components of the final Topic Maps Query Service.

[5]Possible extensions for the Topic Maps Query Service might be additional operations on the Topic Maps data (RESTful access, import/export) or administrative features like caching, security etc.

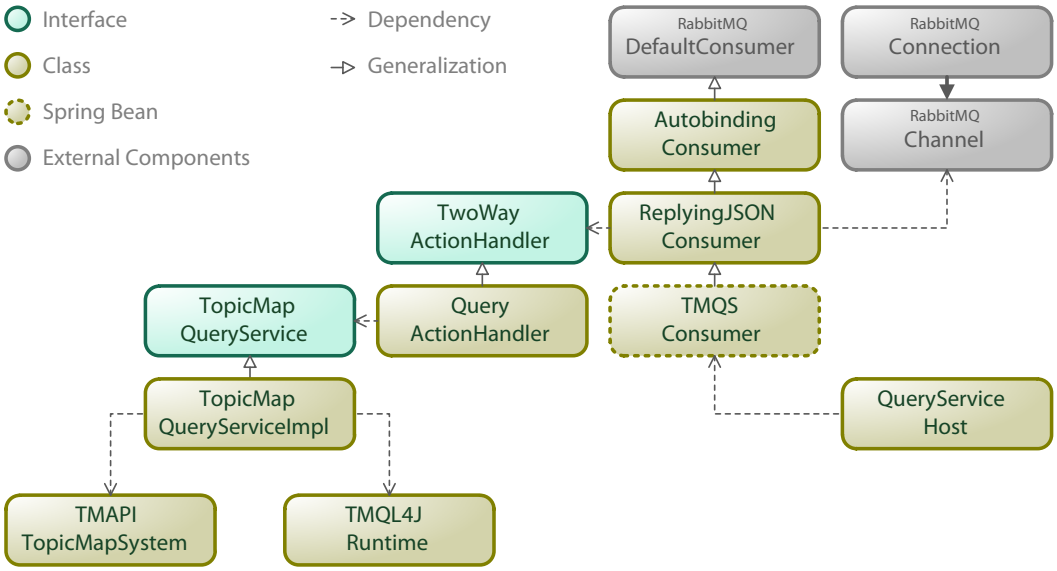

Figure 3.5: Dependencies of the Topic Maps Query Service components.

Besides the `TopicMapQueryServiceImpl` and its underlying `TopicMapQueryService` contract, helper classes were implemented to enable an easy wiring of the standalone service component with the RabbitMQ client library inside of an hosting console application. All required class instances are configured and managed as so-called *Spring Beans*. The `BeanFactory` of the Spring Framework can initialize an *Application Context* from configurations described in .xml files. By using this declarative way and following the *Dependency Injection* pattern, makes it easier to stick together all components.

A future TMQS client has to initialize a reply channel (a separate, exclusive queue) and should provide the generated queue name[6] to the service with each request message. In turn the Topic Maps Query Service itself works as request-response service. Hence the `query` operation is two-way as it receives the query message and returns the converted results of the TMQL4J runtime to the requesting client. To map between the request-response behavior of the `TopicMapQueryServiceImpl` and the message-oriented access layer the `QueryServiceHost` needs to implement:

- a consumer that listens on the query request queue
- a producer that publishes the result on the clients private reply queue

The generic `ReplyingJSONConsumer` is an implementation that combines these two functionalities. Therefore, it requires the injection of an object implementing the `TwoWayActionHandler` interface (like the `QueryActionHandler`) and an instance of a RabbitMQ `Channel`.

[6]The message broker creates a unique name for exclusive, auto-generated message queues.

The `TMQSConsumer` is not a concrete implementation, but represents a bean instance of a `ReplyingJSONConsumer` defined within the `ApplicationContext.xml` . Due to the composition-based approach it does not need any concrete specialization of the base class. As the `TMQSConsumer` instantiates a `ReplyingJSONConsumer` it needs the already discussed `TwoWayActionHandler` instance and a RabbitMQ `Channel` object. Consequently the setup of the whole service host converges at this point to the so-called *composition root*.

```
public interface TwoWayActionHandler<TIn, TOut> {
    public TOut handle(TIn msg) throws Exception;
    public Class<TIn> getRequestMessageType();
    public Class<TOut> getReplyMessageType();
}
```

Listing 3.4: Interface definition for the `QueryActionHandler` implementation.

As the `ReplyingJSONConsumer` needs a `TwoWayActionHandler` an implementation of this interface had to be developed. One concrete descendant of the handler declaration is the `QueryActionHandler` that implements the generic interface with concrete types for `TIn` (=`QueryMessage`) and `TOut` (=`byte[]`). Its constructor expects a `TopicMapQueryService` implementation to be injected when a new instance will be created. Furthermore, the `handle` method must be implemented which finally should extract the relevant information from the incoming `QueryMessage` and invokes the responsible `query` method at the already injected `TopicMapsQueryServiceImpl` instance.

The final console application creates the hosting environment by loading the `ApplicationContext.xml` using the `FileSystemXmlApplicationContext` class from the Spring Framework. The static `main` method allows a single argument to use different .xml files to be used for instantiation. Therefore, multiple bean configurations can be set up to allow a flexible hosting of the same service on different nodes in different environments. The full beans configuration of the application context can be found in Appendix A.5.

```
public class Server {
  public static void main(String[] args) {
    String contextFile = "ApplicationContext.xml";
    if (args.length > 0) contextFile = args[0];

    AbstractApplicationContext ctx =
      new FileSystemXmlApplicationContext(contextFile);
    ctx.registerShutdownHook();
  }
}
```

Listing 3.5: Console application host for the Topic Maps Query Service.

3.3 Benchmarks

By now the message-oriented Topic Maps Query Service solution only supports the TMQL-based `query` operation. Therefore, only the two TMQL operations `TMQLTuples` and `TMQLOccCountFilter` were benchmarked.

Compared to the previous benchmarks in Sections 2.2.1 and 2.3.2 the number of processes increased with each setup by one. Due to the requirement of the additional message broker component the final list of processes contains:

- Benchmarking client application
- RabbitMQ message broker
- Topic Maps Query Service (with instantiated MaJorToM and TMQL4J)
- Redis Server (for the MaJorToM-Redis test case)

All service components shared the same machine for each environment (`local`, `LAN`, `WAN`). Only the benchmarking client application was executed on different nodes except for test cases in the `local` environment.

Operation	MaJorToM-InMemory			MaJorToM-Redis			Requests
	Local	LAN	WAN	Local	LAN	WAN	
◁ TMQLTuples	0.07 s	0.08 s	0.10 s	0.12 s	0.13 s	0.15 s	1
◁ TMQLOccCountFilter	0.35 s	0.34 s	0.41 s	0.45 s	0.46 s	0.52 s	1

* The same pokemon.ltm topic map as in Section 1.2.3 and Section 2.3.2

Table 3.1: Durations for different query operations with TMQS

Referring Table 2.1 a single query obviously takes longer in distributed environments compared to its standalone in-process competitor. This was already expected and visible when discussing the benchmark results of Table 2.2 in Section 2.3.2. The latencies of the message-oriented approach now are related to two phase transmissions (client – message broker – TMQS) and possible internal durations of the message broker. But the results for TMQS show that the querying is faster as its preceding TMoR approach – especially for the LAN or WAN environment. This is the result of the reduction of number of requests needed to get the final query response. The idea behind the TMoR prototype was the almost direct mapping of the TMAPI to an remote contract without a segregated `query` operation. This in turn required an instance of TMQL4J to be initialized on the client side. Therefore, it is parsed and executed locally. Hence each step in the query tree results in one or more remote TMAPI calls using the actual TMoR interfaces. In short: one TMQL query results in one or more TMAPI calls depending on the query statement and the underlying size of the topic map.

With the Topic Maps Query Service the instance of TMQL4J is now located on the server side. Thus a query is passed to the service as-is and only needs a

single request-response pair to be executed. The actual data retrieval against the queried topic map finally results in multiple TMAPI calls which are invoked in-process.

The previous benchmark setup did not cover the possibilities of concurrent querying. To emphasize the scalability features an additional scenario with multiple simultaneous requesting clients was created using an additional server machine.

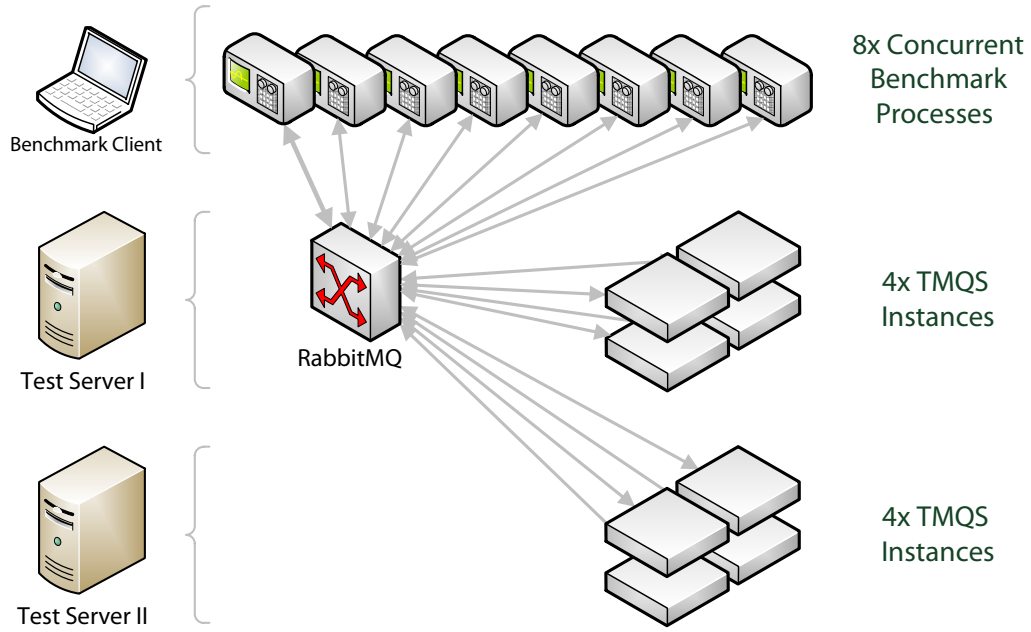

Figure 3.6: Scalability Scenario for Topic Maps Query Service

Extending the number of TMQS processes will increase the amount of possible concurrently running TMQL queries. Figure 3.6 shows a configuration where two quad core servers are hosting one TMQS process per core. This in turn enables eight long lasting TMQL queries to be executed concurrently. As the actual workload is about processing the TMQL query, the overhead for routing the incoming query requests to the final worker process can be ignored. For a further extension of this scenario it only needs the following steps to enhance the service capabilities:

- Attach an additional server machine to the the same network
- Run the required TMQS console instances

3.4 Summary

With the prototype implementation of the Topic Maps Query Service and its benchmarking results the advantages of a message-oriented approach were

shown in this chapter. The use of a middleware tier that supports asynchronous communication and load balancing out of the box allows the configuration of a highly flexible and scalable system.

3.4.1 Extensibility Ideas

As already mentioned in previous sections, one possible extension to enhance the scalability and to show its flexibility can be the implementation of a caching layer. Referring the given architecture it needs an additional implementation of the `TwoWayActionHandler` interface to provide a specialized action handler that internally will not execute the TMQL query, but for instance uses a simple in-memory key value store[7] to realize caching. If the cache does not already contains an entry for the incoming request message, it in turn will forward the query to an responsible exchange point where consumers with an original `TopicMapQueryService` implementation are listening to.

This example can be used to emphasize the flexibility of a message-oriented middleware. Hence the whole implementation, configuration and execution of the additional caching layer can be realized without shutting down the already running productive system with a bunch of Topic Maps Query Service processes. In fact the newly created caching nodes can be added silently.

Another idea will benefit from the routing feature provided by an AMQP-based middle tier. It can be used to configure a weighted load balancing. Focusing on the underlying Topic Maps data it therefore can be possible to group multiple maps to segregated domains. With the use of the AMQP routing features those domains can then be used to configure dynamic forwarding of query request messages to appropriate worker processes as needed. For example: Two queues named `important` and `background` might be established in the message broker whereas seven TMQS worker nodes are listening to the `important` queue, but only one is behind the `background` queue. Hence the load balancing can be controlled by the number of attached processes to the different queues. A list of queued query requests in the `important` queue will now have more CPU time resources (by a factor of seven) compared to queries in the `background` queue.

Thinking of the more complex routing options for *topic* and *headers* exchange types, this method has even more potential to precisely adjust load balancing.

[7] A key value pair for the caching store might be the combination of the hashed query (plus the requested topic map locator) and its JTMQR result.

3.4.2 Limitations and Known Issues of The Current Solution

The given solution in this chapter was focusing on issues for read-only operations. Even if TMQL allows the execution of `ADD` or `UPDATE` statements, the current spike does not care about possible sync problems that may occur where such queries are submitted. In fact there is no transaction management that should prevent inconsistent states of the underlying Topic Maps data. Thus it may lead to an instable system, where single worker nodes may fail. Modifying TMQL queries should therefore be prohibited.

Additionally there is currently no handling of lost connections or any other exception management for the TMQS workers. The main purpose of the presented solution was to proof feasibility and the performance benefits of such an approach.

Next to the known issues about missing query filtering and failure handling, other structural problems exist. The bottom data tier of the whole system still can be the bottleneck when concerning about an possible increase of concurrent queries or the overall data volume of the underlying topic maps. As the query processing worker nodes can be instantiated multiple times, they all have to access the same data source. Hence it requires a unique data provider for all nodes or a distributed solution. Due to a missing synchronization feature between multiple distributed TMQS worker instances an in-memory TMAPI-based Topic Maps engine can therefore not be used as back-end. Referring Table 2.1 the MaJorToM Redis back-end seems to be the only existing choice for this problem. While it is acceptable fast for reading operations, it also supports the distributed access to its databases. But in the scope of this thesis, a measurement for a real world scenario using this integrated approach was not performed.

Chapter 4

Outlook

The previous chapters addressed the accessibility layer for Topic Maps-based back-ends. Hence the attempt to find solutions was made from a top view perspective highlighting distribution technologies for remote access. But focusing on internals of Topic Maps engines in general the message-oriented approach can be used to solve other problems.

4.1 Message-Oriented Approaches for Merging Topic Maps

In fact one primary challenge in the Topic Maps world is the way to implement the merging feature. Hence referring the TMDM, topics with same identifiers are defined to be the same[1] which enables to cross the boundaries of a single topic map. The complexity of detecting merging topics in two topic maps is similar to sort algorithms, as lists of topic identifiers has to be compared with the items in the opposing lists. In a straight forward implementation each topic of one topic map has to be compared with all topics of another topic map.

Two basic approaches for realizing the merging rules within TMAPI-based Topic Maps engines are commonly possible:

Static Merging
: With the merge initiation of two topic maps the comparison and merging process runs until all merging nodes are found. Hence a single merged topic map is the result.

Ad Hoc or Virtual Merging
: The invocation of the merging process does not automatically look up

[1] In addition to the topic equality rules, reifiable constructs are defined to be the same too, if their values and the reifying topic are equal. But for simplicity this rule will be ignored for further discussions.

overlapping topics. Instead the merging happens dynamically when accessing single topics of the virtually merged topic map.

Both approaches have pros and cons. The static merging requires initial processing time and additional memory resources. Referring the read-only TMQS use case, a full merging of the whole topic maps might not be necessary if only e. g. 5% of information is in question for later querying tasks. In addition static merging has bad support for dynamic modifications in topic map sources. Thus it needs some effort to track changes in underlying maps that might effect the merged result. The advantage for such a solution is the responsiveness. It should be as fast as accessing a standard map – obviously the merged map is a topic map.

In turn, a virtually merged mapping needs more resources when accessing the topic map subjects in realtime. The lookup for matching items happens live while reading entities of a map. Introducing a caching layer should reduce occurring latencies.

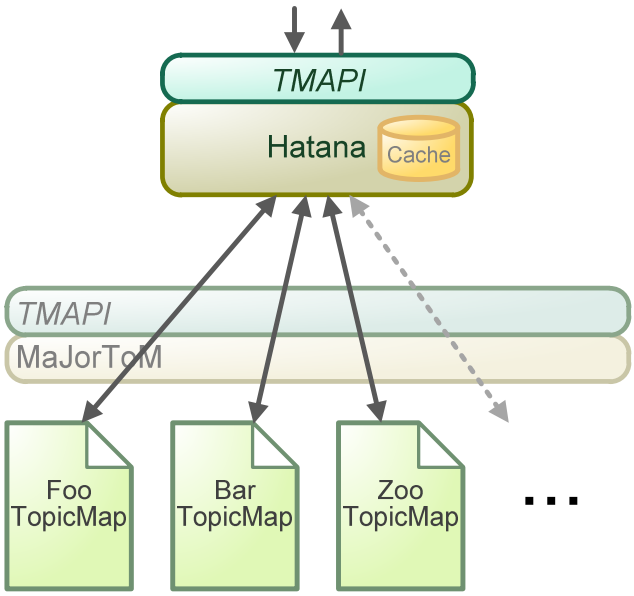

Figure 4.1: Architectural Schema of Hatana.

With Hatana, the Topic Maps Lab provides one implementation for such an ad hoc solution. Speaking in terms of Maiana, those virtual maps are so-called *containers* that are used to manage references to "real" topic maps. Hatana supports caching and implements the TMAPI. It therefore can be utilized as source for other TMAPI-based components like TMQL4J. Ergo, each access on a Topic Maps construct in such an virtual map results in a lookup across all referenced sub topic maps (if the inspected construct is not already cached).

4.2 Merge Registry Service

Using either the static or virtual merging depends on the future use case and the expected environment parameters. For instance:

- Are the source topic maps static or are they modified frequently?
- Will the given hardware provide sufficient resources for the Topic Maps engines and the merging process?
- What is the absolute size (TAO) of the topic map sources?
- Which merging ratios (i. e. merged topics : overall number of topics) are expected?

Thus it might be a good compromise to provide a hybrid approach that supports best of both worlds.

Besides Hatana, one idea for Maianas container feature is initially based on a static merging approach. But instead of creating new topic entities with all merged sub constructs (associations, occurrences, names and variants), the merging is only concentrated on the actual topic identifiers to prevent duplication of data. This method will keep the original source topic maps untouched and will only persist the merge information (i. e. the references of the merging topics). Furthermore, the merge process will run asynchronously in the background after a container was modified (i. e. adding or removing topic map references). The use of message-oriented middleware fits in this scenario to enable access to the merge information and to send notification about changes in the source topic maps by the underlying Topic Maps engines.

```
public interface MergeRegistryService
{
  void registerTopicMap(String topicMapLocator,
                        List<String[]> identifiers);
  void unregisterTopicMap(String topicMapLocator);

  int createContainer(String[] tmLocators);
  void destroyContainer(int containerId);
  bool isContainerMerging(int containerId);

  MergedTopics findInContainer(int containerId,
                               String[] identifiers);
}

public class MergedTopics extends HashMap<String, String[]> {}
```

Listing 4.1: Contracts describing a possible `MergeRegistryService` and its request result data structure.

The runtime implementation of a Merge Registry Service might define methods as presented with `MergeRegistryService` interface in Listing 4.1 to handle the following three core operation phases:

Topic map registration

This initial step is required to inform the Merge Registry Service about related topic maps. As only the subject identifiers, subject locators and item identifiers of topics are relevant for merging, simple lists of string arrays containing the URIs cope with the needs. Hence not the full topic map will be submitted for an registration, but a list of all topic identifiers grouped by topic[2]. Listing 4.2 shows the JSON representation of a small sample topic map `tm_foo` containing only the identifiers of two topics `a` and `b`.

```
{ "http://example.org/tm_foo":[ // topic map 'Foo'
    ["si:a", "ii:x"],           // topic 'a'
    ["si:b"]                    // topic 'b'
  ]}
```

Listing 4.2: Submitted list of topic identifiers for registration in Merge Registry Service.

All registered topic maps and their containing topic identifiers are stored in a key-value store using the topic maps locator URI as key. Listing 4.3 depicts the structure in a JSON sample.

[2]To distinguish between the identifier types all representations need to be prefixed with corresponding abbreviations as defined for the JTM format: `si:`, `sl:` or `ii:`.

```
{ "http://example.org/tm_foo":[
    ["si:a", "ii:x"], // topic 'a'
    ["si:b"]          // topic 'b'
  ],
  "http://example.org/tm_bar":[
    ["si:a", "si:b"]  // bridging topic 'a' and 'b'
    ["si:c"]          // topic 'c'
  ]}
```

Listing 4.3: Storage of topic map references and their topic identifier lists in Merge Registry Service.

Container management

By defining a container using references to the previously registered Topic Maps (i. e. `tm_foo` and `tm_bar`) the merging can start asynchronously in the background. Within the merge process the provided `findInContainer` method already enables the lookup of partially merged topic identifier lists. This feature in turn can be used to include intermediate data for queries against containers that are currently merging. Invoking `isContainerMerged` can be used to get the current merging status of a container for front-end notification.

The structure for the internal storage of the merge information can be realized as key-value dictionary too. Given a unified list of the topic identifiers of all topic maps that are referenced in the container, each identifier will be used as key addressing an array of the merged topic identifiers. Listing 4.4 shows a possible structure to store container references and the resulting merge index.

```
{ "merged_foo_and_bar":{
    "references":[
      "http://example.org/tm_foo",
      "http://example.org/tm_bar"],
    "merged_index":[
      "si:a":["si:a", "ii:x", "si:b"],
      "ii:x":["si:a", "ii:x", "si:b"],
      "si:b":["si:a", "ii:x", "si:b"],
      "si:c":["si:c"]
    ]},
  "empty_container":{
     "references":[],
     "merged_index":[]
  }}
```

Listing 4.4: JSON representation of container definitions and the actual merge registry in the Merge Registry Service.

Query transformation

For the remaining functionality a specialized TMQL engine needs to be implemented. It should use the merge knowledge, provided by the Merge Registry Service if queries were executed targeting containers. Internally the query will then be translated by replacing the involved topic identifiers in the query string with the list of merged identifiers. Those modified statements in turn can be used to query all affected topic maps concurrently using the distributed `TopicMapsQueryService`. Finally the returned results have to be merged again before creating the JTMQR response message for the client. Figure 4.2 shows a possible workflow of a query operation that uses a container as data source.

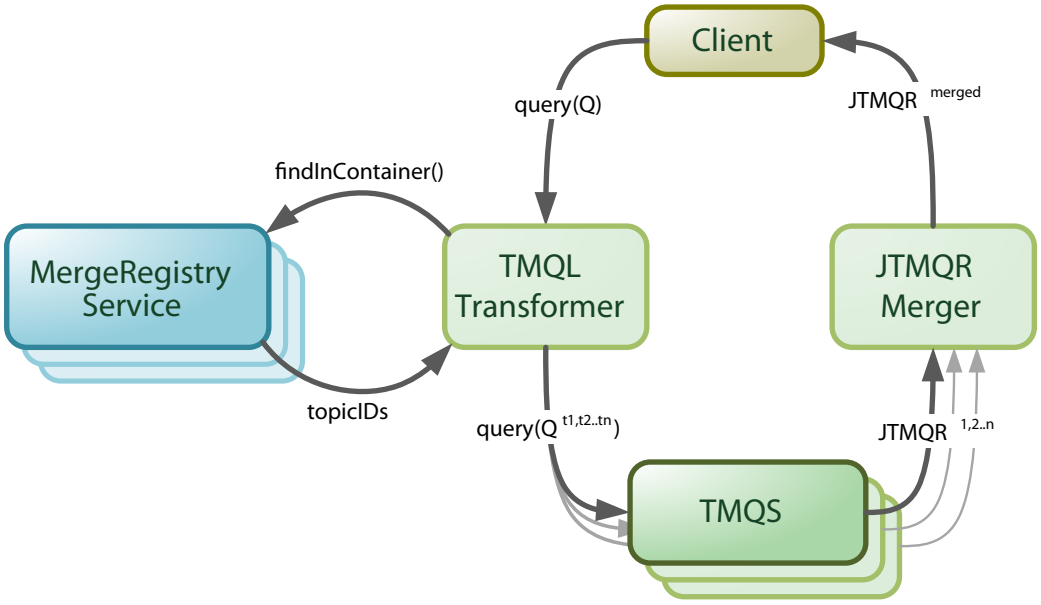

Figure 4.2: Workflow of query processing for the Merge Registry Service.

As already shown with TMQS in Section 3.2, a distributed configuration of multiple concurrent working `MergeRegistryService` nodes using message-oriented middleware seems to be a possible way to enable load balanced scalability for the merge and query process.

4.3 Distributed TMQL Processing on Merged Topic Maps

As the querying against containers requires the merge information, it can benefit from the previous idea of extracting the merge process into a separate service. Assuming *all* service nodes have access to the same shared Topic Maps data, internal parts of the query process can be parallelized too.

The most common way to implement a query language processor is to follow the ideas of compilers. It requires initial lexical analysis and parsing to achieve a stage where the binding to an actual processing component can be established. In fact TMQL4J follows this path and internally produces a so called *expression tree* that provides a hierarchical structure (see `ParserTreeImpl` class) reflecting the canonicalized query statement. After the tree was built an interpreter walks along the tree, retrieving the requested data from the underlying topic map for each tree node to finally create the `IResultSet` instance. A sample tree for a TMQL query can be found in Appendix A.6.

After inspecting the internals of TMQL4J, the entry point to create a custom expression tree interpreter was found in the `ParserTreeImpl`. This enables the distribution of parted sub query trees by developing a specialized `IExpressionInterpreter`.

Referring Appendix A.6, the new interpreter should be focused on tree nodes like `Anchor` and `Step` that are actually related to sets of topics. Hence these nodes require a lookup in the Merge Registry Service to find matching merge information about the same subject in different maps. If a topic occurs in multiple maps, a parallel sub querying on lower parts of the query expression tree can be triggered. The method of spawning sub query process can is as recursive way to distribute the workload.

Each sub query worker obviously needs information about the current state of the query process. In fact a serialized representation of the query expression tree with the possibility to address the current node in progress are required.[3] In turn, the child processors can now execute the sub queries using this information and the required access to topic maps and the merge data.

[3]For rapid prototyping the whole query can simply be submitted as full statement to be lexed and parsed again in the sub query node.

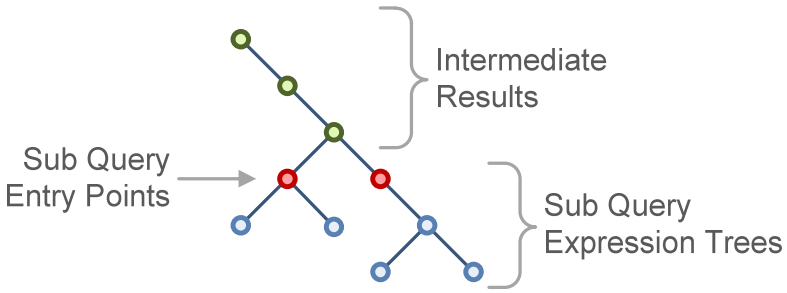

Figure 4.3: Data required for Sub Querying: the Query Expression Tree itself, a pointer to address the sub query entry point and possible intermediate results of previous sub queries.

In addition, already achieved intermediate results and global information (e. g. prefixes defined in the statement) should be accessible to sub processes handling query fragments. Literally the proposed data structure for sub queries requires additional fields to provide the intermediate results. Depending on the complexity of the queries and the – more important – size of a container and its referenced topic maps the intermediate results can grow quickly. Hence it might be useful to share those temporary information with specialized infrastructure like Redis instead of pushing those data as message overhead between the multiple query processor nodes each time.

The distribution of sub queries furthermore requires correlation management in the parent nodes to track the dependencies between a sub query and its origin. Using a concept of tasks might be helpful to gain asynchronous behavior for the multiple sub queries, similar to the *MapReduce* approach. In detail, stacks can be used to keep the knowledge about running jobs. If the list of tasks is cleared, the distribution of sub queries becomes synchronized. Hence the prepared result of all sub queries can be returned to the parent requesting node – that finally is the client. But before returning any results to the parent node, the sub results may need to be processed. This includes merging, filtering, aggregation or simply bypassing all sub results to the parent. Figure 4.4 depicts the message flow for a sample query.

All in all, such architecture requires query nodes to be implemented as root *and* sub query processors in a single component. Due to its asynchronous approach this enables instances to manage sub queries from different root nodes while waiting for other processes to complete tasks of another query. Hence an inherent load balancing can be achieved easily by using message-oriented middleware as central distribution infrastructure.

As this proposal is just an theoretical idea there is no concrete implementation of a prototype available at time of writing.

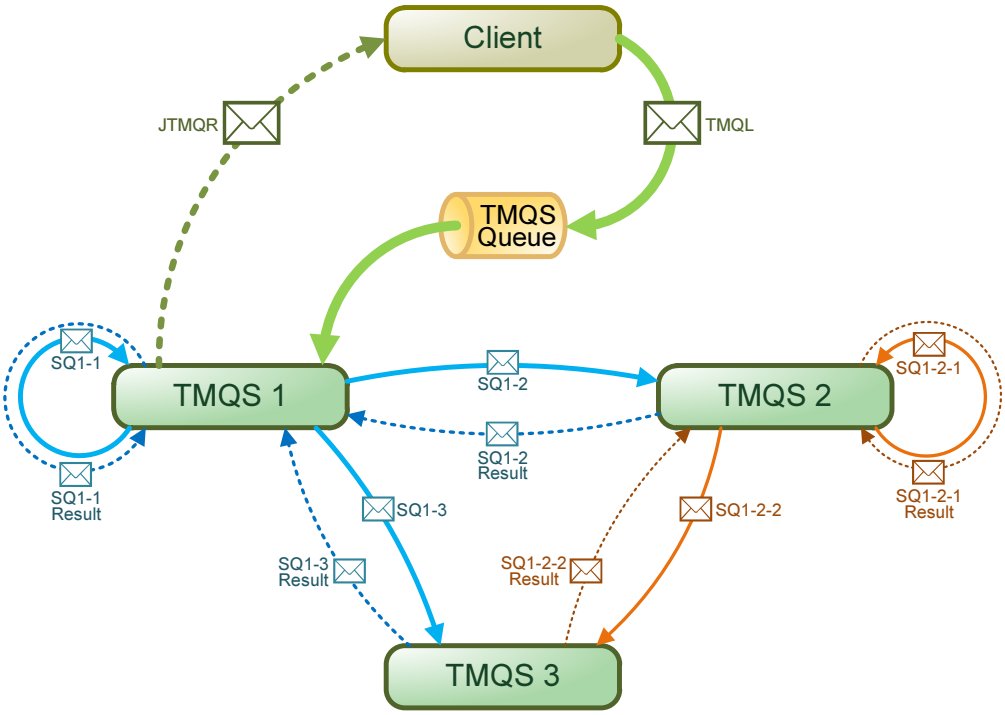

Figure 4.4: Sample of sub querying with extended Topic Maps Query Service instances. The initial TMQL statement from the client will be divided into three sub queries `SQ1-1`, `SQ1-2`, `SQ1-3` and distributed to the three TMQS nodes[4]. In turn `TMQS 2` receives `SQ1-2` and will divide it again into `SQ1-2-1` and `SQ1-2-2`. After all results were sent back the initial root `TMQS 1` the final JTMQR response message will be created and returned to the client.

4.4 Combining Presented Approaches

The Topic Maps Query Service with an extension to support distributed sub querying now fits into a larger scenario as shown in Figure 4.5. This configuration can be built upon multiple instances of Merge Registry Service and elementary Topic Maps Management Services next to the TMQS workers. With the use of Redis (for topic maps data persistence and the additional merge information) a sharable persisting basis is necessary to provide all services with the same consistent data.

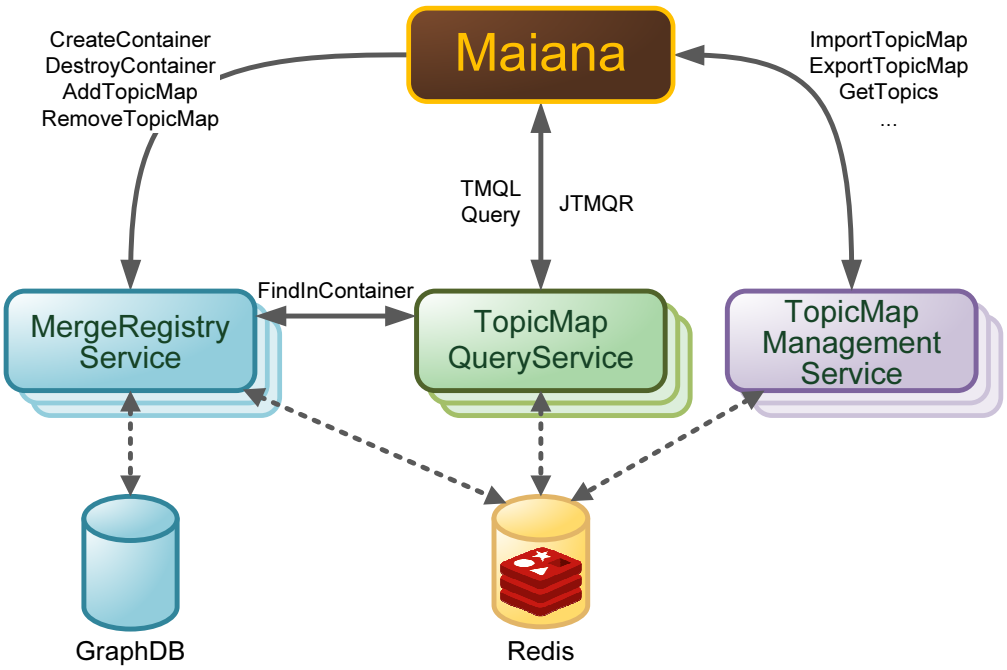

Figure 4.5: Architectural schema of a distributed Topic Maps Query Service supporting partial sub querying.

The given arrangement of services is a flexible system with support of easy scalability as it is build on a message-oriented architecture with well-defined message contracts for communication. Sticking all mentioned approaches together, an ideal workflow sequence for such a system will then consist of the following three phases:

Manage Topic Maps
: Suppling shared topic maps data using an import service to enable the upload of topic maps in common data formats (XTM, LTM, JTM, etc.). Those can be accessed afterwards from clients directly using RESTful operations. Using JTM or XTM fragments as serialization format enables the partial access to the content of the maps.

Manage Containers
: Create (and delete) virtual merge containers that allow adding and removing references to previously imported maps. Each change in a container may automatically trigger an update in the merged virtually in a background process. This task extracts and saves the significant merge information into a shared data store (e. g. a graph database). The merging can be processed in parallel as described in Section 4.2.

Querying Maps and Containers
: A set of query services are consuming queues that can be filled with query request messages from clients (as shown in Section 3.2). A query can either address a single map or a virtual container. For containers the specialized processor described in previous Section 4.3 can produce sub queries in a distributed manner.

This certainly needs some tweaks and finally is not implemented by now, but can be seen as basis for a robust and flexible solution.

Chapter 5

Conclusion

Using message-oriented middleware provides a flexible environment to create scalable systems. The underlying concept of queues, producers and consumers enforces a component-oriented thinking using contract-first principles when designing the architecture for an data intensive service application like Maiana. Message brokers like RabbitMQ are easy to configure and enable cross-platform access to a messaging system by providing different client libraries for developers. This thesis showed how to use these concepts to improve computationally intensive operations in Topic Maps applications. In the first place the data extraction by running queries against topic maps can be parallelized and therefore be scaled. This can be achieved by exposing multiple Topic Maps Query Service worker processes on one or more machines that are connected to the same message broker queue listening for query request messages. The next idle worker will receive and process the request until the queue is empty.

Based on this technique of distributing decoupled query units, other features of the Topic Maps world can be solved. In fact the Outlook chapter presented approaches to handle the Topic Maps merging problem. By extracting the actual process into a dedicated Merge Registry Service, the search for merging topics can be executed concurrently. Furthermore, it shows that already found results can be used immediately while the merging is still in progress.

Attaching more machines to the existing network to enable spawning of new worker processes, is a typical scale-out procedure. Looking at today's cloud computing services those loosely coupled worker implementations are the basis when thinking of a bigger, global idea. Hence a dynamically workload-dependent scaling of a such a service seems to be feasible.

At the moment of writing, the underlying TMAPI-based back-end bindings (using the RTM libraries) of Maiana were already replaced with a TMQL-based access layer. Even though not all features of Maiana can be realized using a pure TMQL solution by now, a first step to decouple the monolithic TMAPI stack and to enable flexibility and scalability was made.

Furthermore, this work depicted essential conceptual issues in the scope of service-oriented architectures. Distributed systems need a different point of view when thinking of interfaces and data transmission. For instance, TMAPI should *not* be used as contract for remote operations. The interface is more or less reflecting a data structure and defines methods to access elementary Topic Maps data. Anyway, TMAPI is still a good choice for internal atomic Topic Maps features in the scope of a single client-side process. Additionally the JTMQR data structure was developed in this work to enable the transport of TMQL results back to the client. It is wrapping the JSON Topic Maps format to allow submission of single Topic Maps constructs.

Finally, the discussed approaches do not represent a generic solution for *all* Topic Maps-related performance issues. They mainly target the scalability and concurrency problems that actually occur within the Maiana web application. This thesis assumed that many heterogeneous, independent topic maps coexist in the same environment, whereas multiple consumers need read access to fragments of those data pools. In fact not all performance issues could be solved within this thesis. The bottom layer, responsible for persisting and sharing the raw Topic Maps data is still based on the Topic Maps Data Model (see [TMD08]). Using such relational normalization, the retrieval of constructs requires multiple database queries. Hence a renewal of the data layer is needed to complete the concept of distribution. Nevertheless, the prototypes presented in this thesis used MaJorToM-Redis, as it represents a persisting Topic Maps back-end with acceptable performance.

Overall, the results of this work can be seen as basic concept for further implementations that efficiently use a message-oriented architecture to create scalable Topic Maps services.

Bibliography

[AFG+09] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. Above the Clouds : A Berkeley View of Cloud Computing Cloud Computing : An Old Idea Whose Time Has (Finally) Come. *Computing*, pages 07–013, 2009.

[Ahm09] Kal Ahmed. Making Topic Maps SPARQL. In *Proceedings of the TMRA 2009 - Linked Topic Maps*, 2009.

[Bar05] Robert A Barta. TMIP, A RESTful Topic Maps Interaction Protocol. In *Extreme Markup Languages*, 2005.

[Bar09] Getting started with AMQP and RabbitMQ. http://www.infoq.com/articles/AMQP-RabbitMQ, 2009. [Online; accessed 8-February-2011].

[BDHM08] Khalid Belhajjame, Mathieu D'Aquin, Peter Haase, and Paolo Missier, editors. *First International Workshop on Semantic Metadata Management and Applications, SeMMA 2008, Located at the Fifth European Semantic Web Conference (ESWC 2008), Tenerife, Spain, June 2nd, 2008. Proceedings*, volume 346 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2008.

[Cer10] Robert Cerny. JSON Topic Maps 1.1. http://www.cerny-online.com/jtm/1.1/, 2010. [Online; accessed 24-March-2011].

[DS10] Haig Djambazian and Rob Sladek. Case Study of Scientific Data Processing on a Cloud Using Hadoop. *Data Processing*, pages 400–415, 2010.

[EML05] *Proceedings of the Extreme Markup Languages 2005 Conference, 1-5 August 2005, Montr{é}al, Quebec, Canada*, 2005.

[Fie00] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, UNIVERSITY OF CALIFORNIA, IRVINE, 2000.

[Fow02] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional, 2002.

[Gar05] Lars Marius Garshol. tolog - A Topic Maps Query Language. In Lutz Maicher and Jack Park, editors, *Proceedings of the TMRA 2005*, volume 3873 of *Lecture Notes in Computer Science*, pages 183–196. Springer, 2005.

[GB06] Lars Marius Garshol and Dmitry Bogachev. TM / XML – Topic Maps Fragments in XML, 2006.

[Gre08] Daniel Gredler. Java Remoting: Protocol Benchmarks. http://daniel.gredler.net/2008/01/07/java-remoting-protocol-benchmarks/, 2008. [Online; accessed 15-July-2011].

[HW03] Gregor Hohpe and Bobby Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley Professional, 2003.

[KHK10] Yuuki Kuribara, Takeshi Hosoya, and Masaomi Kimura. TOME: The Topic Maps Database Extended. In *Proceedings of SEATUC Symposium*, pages 245–248, 2010.

[KK10] Yuki Kuribara and Masaomi Kimura. Inquiry Optimization Technique for a Topic Map Database. In *Proceedings of TMRA 2010 - Information wants to be a Topic Map*, pages 53–62, 2010.

[LK09] Hyun Jung La and Soo Dong Kim. A Systematic Process for Developing High Quality SaaS Cloud Services *. *Development*, pages 278–289, 2009.

[MP06] Lutz Maicher and Jack Park, editors. *Charting the Topic Maps Research and Applications Landscape, First International Workshop on Topic Maps Research and Applications, TMRA 2005, Leipzig, Germany, October 6-7, 2005, Revised Selected Papers*, volume 3873 of *Lecture Notes in Computer Science*. Springer, 2006.

[Par07] Terence Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. Pragmatic Programmers. Pragmatic Bookshelf, first edition, May 2007.

[SR08] Silvia Stefanova and Tore Risch. Viewing and Querying Topic Maps in terms of RDF. In Khalid Belhajjame, Mathieu D'Aquin, Peter Haase, and Paolo Missier, editors, *SeMMA*, volume 346 of *CEUR Workshop Proceedings*, pages 69–83. CEUR-WS.org, 2008.

[Tan02] Andrew S. Tanenbaum. *Computer Networks (4th Edition)*. Prentice Hall, 2002.

[TMD08] Topic Maps – Data Model. http://www.isotopicmaps.org/sam/sam-model/, 2008. [Online; accessed 3-April-2011].

[WRW96] Ann Wollrath, Roger Riggs, and Jim Waldo. A Distributed Object Model for the Java System A Distributed Object Model for the Java ™ System. In *Proceedings of the USENIX 1996*, number June, 1996.

[ZL08] Jiangong Zhang and Xiaohui Long. Performance of Compressed Inverted List Caching in Search Engines. *Byte*, pages 387–396, 2008.

Appendix A

Appendix

A.1 MaJorToM PostgreSQL Log Analysis Report (Before Optimization)

See also: MaJorToM issue 68

pgFouine: PostgreSQL log analysis report

Overall statistics | Queries by type | Queries that took up the most time (N) | Slowest queries | Most frequent queries (N) | Slowest queries (N)

Normalized reports are marked with a "(N)".

- Generated on 2010-10-25 15:48
- Parsed C:\Program Files (x86)\PostgreSQL\8.4\data\pg_log\postgresql-2010-10-25.log (16,115 lines) in 30s
- Log from 2010-10-25 15:46:31 to 2010-10-25 15:47:39

Overall statistics ^

- Number of unique normalized queries: 40
- Number of queries: 4,007
- Total query duration: 5.8s
- First query: 2010-10-25 15:46:31
- Last query: 2010-10-25 15:47:39
- Query peak: 1,112 queries/s at 2010-10-25 15:46:40

Queries by type ^

Type	Count	Percentage
SELECT	3,851	96.1
INSERT	150	3.7

Queries that took up the most time (N) ^

Rank	Total duration	Times executed	Av. duration (s)	Query
1	**2.2s**	241	0.01	`SELECT id_type FROM typeables AS ty, topics AS t WHERE ty.id = 0;` Show examples
2	**0.7s**	369	0.00	`SELECT id FROM names WHERE id_parent = 0 AND id_scope = 0;` Show examples
3	**0.6s**	104	0.01	`SELECT id, id_parent FROM roles WHERE id_player = 0;` Show examples
4	**0.4s**	109	0.00	`SELECT l.id,reference FROM locators AS l, rel_item_identifiers AS r WHERE r.id_construct = 0 AND r.id_locator = l.id;` Show examples
5	**0.4s**	206	0.00	`SELECT id FROM names WHERE id_parent = 0;` Show examples
6	**0.3s**	112	0.00	`SELECT t.id FROM topics AS t, locators AS l, rel_subject_identifiers AS r WHERE id_topicmap = 0 AND reference = '' AND l.id = r.id_locator AND r.id_topic = t.id;` Show examples
7	**0.3s**	59	0.00	`SELECT id FROM roles WHERE id_parent = 0;` Show examples
8	**0.3s**	155	0.00	`SELECT l.id,reference FROM locators AS l, rel_subject_identifiers AS r, topics AS t WHERE r.id_topic = 0 AND r.id_locator = l.id AND t.id = r.id_topic;` Show examples

9	**0.2s**	1,984	0.00	SELECT DISTINCT id_theme FROM rel_themes WHERE id_scope = 0; Show examples
10	**0.2s**	103	0.00	SELECT id FROM occurrences WHERE id_parent = 0; Show examples
11	**0.1s**	51	0.00	SELECT id FROM names WHERE id_parent = 0 AND id_type = 0; Show examples
12	**0.0s**	8	0.01	SELECT id, id_parent FROM roles WHERE id_player = 0 AND id_type = 0; Show examples
13	**0.0s**	22	0.00	SELECT id_scope FROM names WHERE id_topicmap = 0; Show examples
14	**0.0s**	6	0.00	WITH iis AS (SELECT id_construct FROM rel_item_identifiers, locators WHERE id = id_locator AND reference = '')SELECT id, id_parent, 0 AS other, '' AS type FROM topics WHERE id IN (SELECT id_construct FROM iis) AND id_topicmap = 0 UNION SELECT id, id_parent, 0 AS other, '' AS type FROM associations WHERE id IN (SELECT id_construct FROM iis)AND id_topicmap = 0 UNION SELECT id, id_parent, 0 AS other, '' AS type FROM names WHERE id IN (SELECT id_construct FROM iis) AND id_topicmap = 0 UNION SELECT id, id_parent, 0 AS other, '' AS type FROM occurrences WHERE id IN (SELECT id_construct FROM iis) AND id_topicmap = 0 UNION SELECT v.id, v.id_parent, n.id_parent, '' AS type FROM variants AS v, names AS n WHERE v.id IN (SELECT id_construct FROM iis) AND v.id_parent = n.id AND v.id_topicmap = 0UNION SELECT id, id_parent, 0 AS other, '' AS type FROM roles WHERE id IN (SELECT id_construct FROM iis) AND id_topicmap = 0 UNION SELECT id, 0 AS id_parent, 0 AS other, '' AS type FROM topicmaps WHERE id IN (SELECT id_construct FROM iis); Show examples
15	**0.0s**	146	0.00	INSERT INTO locators (reference) SELECT '' WHERE NOT EXISTS (SELECT id FROM locators WHERE reference = '') RETURNING *; Show examples
16	**0.0s**	1	0.03	SELECT DISTINCT id_type FROM rel_instance_of, topics WHERE id_topicmap = 73060 AND id = id_type AND id_instance IN (SELECT id FROM topics AS t WHERE t.id = id_instance);
17	**0.0s**	52	0.00	SELECT n.nspname,c.relname,a.attname,a.atttypid,a.attnotnull,a.atttypmod,a.attlen,a.attnum,pg_catalog.pg_get_expr(def.adbin, def.adrelid) AS adsrc,dsc.description,t.typbasetype,t.typtype FROM pg_catalog.pg_namespace n JOIN pg_catalog.pg_class c ON (c.relnamespace = n.oid) JOIN pg_catalog.pg_attribute a ON (a.attrelid=c.oid) JOIN pg_catalog.pg_type t ON (a.atttypid = t.oid) LEFT JOIN pg_catalog.pg_attrdef def ON (a.attrelid=def.adrelid AND a.attnum = def.adnum) LEFT JOIN pg_catalog.pg_description dsc ON (c.oid=dsc.objoid AND a.attnum = dsc.objsubid) LEFT JOIN pg_catalog.pg_class dc ON (dc.oid=dsc.classoid AND dc.relname='') LEFT JOIN pg_catalog.pg_namespace dn ON (dc.relnamespace=dn.oid AND dn.nspname='') WHERE a.attnum > 0 AND NOT a.attisdropped AND c.relname LIKE '' ORDER BY nspname,relname,attnum; Show examples
18	**0.0s**	4	0.00	SELECT id FROM roles WHERE id_parent = 0 AND id_type = 0; Show examples
19	**0.0s**	92	0.00	SELECT DISTINCT value FROM literals WHERE id = 0; Show examples
20	**0.0s**	1	0.01	SELECT DISTINCT id_type FROM occurrences WHERE id_topicmap = 73060;

Slowest queries ^

Rank	Duration (s)	Query
1	**0.03**	SELECT DISTINCT id_type FROM rel_instance_of, topics WHERE id_topicmap = 73060 AND id = id_type AND id_instance IN (SELECT id FROM topics AS t WHERE t.id = id_instance);
2	**0.02**	SELECT id_type FROM typeables AS ty, topics AS t WHERE ty.id = 73459;
3	**0.01**	SELECT id_type FROM typeables AS ty, topics AS t WHERE ty.id = 73441;
4	**0.01**	SELECT id_type FROM typeables AS ty, topics AS t WHERE ty.id = 73184;
5	**0.01**	SELECT id_type FROM typeables AS ty, topics AS t WHERE ty.id = 73329;
6	**0.01**	SELECT id, id_parent FROM roles WHERE id_player = 73097 AND id_type = 73165;
7	**0.01**	SELECT id_type FROM typeables AS ty, topics AS t WHERE ty.id = 73423;
8	**0.01**	SELECT id_type FROM typeables AS ty, topics AS t WHERE ty.id = 73314;
9	**0.01**	SELECT id_type FROM typeables AS ty, topics AS t WHERE ty.id = 73338;
10	**0.01**	SELECT id_type FROM typeables AS ty, topics AS t WHERE ty.id = 73168;
11	**0.01**	SELECT id_type FROM typeables AS ty, topics AS t WHERE ty.id = 73166;
12	**0.01**	SELECT id_type FROM typeables AS ty, topics AS t WHERE ty.id = 73283;
13	**0.01**	SELECT id_type FROM typeables AS ty, topics AS t WHERE ty.id = 73386;
14	**0.01**	SELECT id_type FROM typeables AS ty, topics AS t WHERE ty.id = 73382;
15	**0.01**	SELECT id_type FROM typeables AS ty, topics AS t WHERE ty.id = 73289;
16	**0.01**	SELECT id_type FROM typeables AS ty, topics AS t WHERE ty.id = 73251;
17	**0.01**	SELECT id_type FROM typeables AS ty, topics AS t WHERE ty.id = 73261;
18	**0.01**	SELECT id_type FROM typeables AS ty, topics AS t WHERE ty.id = 73263;
19	**0.01**	SELECT id_type FROM typeables AS ty, topics AS t WHERE ty.id = 73266;
20	**0.01**	SELECT id_type FROM typeables AS ty, topics AS t WHERE ty.id = 73485;

Most frequent queries (N) ^

Rank	Times executed	Total duration	Av. duration (s)	Query
1	**1,984**	0.2s	0.00	SELECT DISTINCT id_theme FROM rel_themes WHERE id_scope = 0; Show examples
2	**369**	0.7s	0.00	SELECT id FROM names WHERE id_parent = 0 AND id_scope = 0; Show examples
3	**241**	2.2s	0.01	SELECT id_type FROM typeables AS ty, topics AS t WHERE ty.id = 0; Show examples
4	**206**	0.4s	0.00	SELECT id FROM names WHERE id_parent = 0; Show examples

5	**155**	0.3s	0.00	SELECT l.id,reference FROM locators AS l, rel_subject_identifiers AS r, topics AS t WHERE r.id_topic = 0 AND r.id_locator = l.id AND t.id = r.id_topic; Show examples
6	**146**	0.0s	0.00	INSERT INTO locators (reference) SELECT '' WHERE NOT EXISTS (SELECT id FROM locators WHERE reference = '') RETURNING *; Show examples
7	**112**	0.3s	0.00	SELECT t.id FROM topics AS t, locators AS l, rel_subject_identifiers AS r WHERE id_topicmap = 0 AND reference = '' AND l.id = r.id_locator AND r.id_topic = t.id; Show examples
8	**111**	0.0s	0.00	SELECT l.id,reference FROM locators AS l, rel_subject_locators AS r , topics AS t WHERE r.id_topic = 0 AND r.id_locator = l.id AND t.id = r.id_topic; Show examples
9	**109**	0.4s	0.00	SELECT l.id,reference FROM locators AS l, rel_item_identifiers AS r WHERE r.id_construct = 0 AND r.id_locator = l.id; Show examples
10	**104**	0.6s	0.01	SELECT id, id_parent FROM roles WHERE id_player = 0; Show examples
11	**103**	0.2s	0.00	SELECT id FROM occurrences WHERE id_parent = 0; Show examples
12	**92**	0.0s	0.00	SELECT DISTINCT value FROM literals WHERE id = 0; Show examples
13	**59**	0.3s	0.00	SELECT id FROM roles WHERE id_parent = 0; Show examples
14	**52**	0.0s	0.00	SELECT n.nspname,c.relname,a.attname,a.atttypid,a.attnotnull,a.atttypmod,a.attlen,a.attnum,pg_catalog.pg_get_expr(def.adbin, def.adrelid) AS adsrc,dsc.description,t.typbasetype,t.typtype FROM pg_catalog.pg_namespace n JOIN pg_catalog.pg_class c ON (c.relnamespace = n.oid) JOIN pg_catalog.pg_attribute a ON (a.attrelid=c.oid) JOIN pg_catalog.pg_type t ON (a.atttypid = t.oid) LEFT JOIN pg_catalog.pg_attrdef def ON (a.attrelid=def.adrelid AND a.attnum = def.adnum) LEFT JOIN pg_catalog.pg_description dsc ON (c.oid=dsc.objoid AND a.attnum = dsc.objsubid) LEFT JOIN pg_catalog.pg_class dc ON (dc.oid=dsc.classoid AND dc.relname='') LEFT JOIN pg_catalog.pg_namespace dn ON (dc.relnamespace=dn.oid AND dn.nspname='') WHERE a.attnum > 0 AND NOT a.attisdropped AND c.relname LIKE '' ORDER BY nspname,relname,attnum; Show examples
15	**51**	0.1s	0.00	SELECT id FROM names WHERE id_parent = 0 AND id_type = 0; Show examples
16	**37**	0.0s	0.00	SELECT id_reifier FROM reifiables WHERE id = 0; Show examples
17	**22**	0.0s	0.00	SELECT id_scope FROM names WHERE id_topicmap = 0; Show examples
18	**8**	0.0s	0.01	SELECT id, id_parent FROM roles WHERE id_player = 0 AND id_type = 0; Show examples
19	**6**	0.0s	0.00	WITH iis AS (SELECT id_construct FROM rel_item_identifiers, locators WHERE id = id_locator AND reference = '')SELECT id, id_parent, 0 AS other, '' AS type FROM topics WHERE id IN (SELECT id_construct FROM iis) AND id_topicmap = 0 UNION SELECT id, id_parent, 0 AS other, '' AS type FROM associations WHERE id IN (SELECT id_construct FROM iis)AND id_topicmap = 0 UNION SELECT id, id_parent, 0 AS other, '' AS type FROM names WHERE id IN (SELECT id_construct FROM iis) AND id_topicmap = 0 UNION SELECT id, id_parent, 0 AS other, '' AS type FROM occurrences WHERE id IN (SELECT id_construct FROM iis) AND id_topicmap = 0 UNION SELECT v.id, v.id_parent, n.id_parent, '' AS type FROM variants AS v, names AS n WHERE v.id IN (SELECT id_construct FROM iis) AND v.id_parent = n.id AND v.id_topicmap = 0UNION SELECT id, id_parent, 0 AS other, '' AS type FROM roles WHERE id IN (SELECT id_construct FROM iis) AND id_topicmap = 0 UNION SELECT id, 0 AS id_parent, 0 AS other, '' AS type FROM topicmaps WHERE id IN (SELECT id_construct FROM iis); Show examples
20	**4**	0.0s	0.00	SELECT id FROM roles WHERE id_parent = 0 AND id_type = 0; Show examples

Slowest queries (N) ^

Rank	Av. duration (s)	Times executed	Total duration	Query
1	**0.03**	1	0.0s	SELECT DISTINCT id_type FROM rel_instance_of, topics WHERE id_topicmap = 73060 AND id = id_type AND id_instance IN (SELECT id FROM topics AS t WHERE t.id = id_instance);
2	**0.01**	241	2.2s	SELECT id_type FROM typeables AS ty, topics AS t WHERE ty.id = 0; Show examples
3	**0.01**	8	0.0s	SELECT id, id_parent FROM roles WHERE id_player = 0 AND id_type = 0; Show examples
4	**0.01**	104	0.6s	SELECT id, id_parent FROM roles WHERE id_player = 0; Show examples
5	**0.01**	1	0.0s	SELECT DISTINCT id_type FROM occurrences WHERE id_topicmap = 73060;
6	**0.00**	59	0.3s	SELECT id FROM roles WHERE id_parent = 0; Show examples
7	**0.00**	6	0.0s	WITH iis AS (SELECT id_construct FROM rel_item_identifiers, locators WHERE id = id_locator AND reference = '')SELECT id, id_parent, 0 AS other, '' AS type FROM topics WHERE id IN (SELECT id_construct FROM iis) AND id_topicmap = 0 UNION SELECT id, id_parent, 0 AS other, '' AS type FROM associations WHERE id IN (SELECT id_construct FROM iis)AND id_topicmap = 0 UNION SELECT id, id_parent, 0 AS other, '' AS type FROM names WHERE id IN (SELECT id_construct FROM iis) AND id_topicmap = 0 UNION SELECT id, id_parent, 0 AS other, '' AS type FROM occurrences WHERE id IN (SELECT id_construct FROM iis) AND id_topicmap = 0 UNION SELECT v.id, v.id_parent, n.id_parent, '' AS type FROM variants AS v, names AS n WHERE v.id IN (SELECT id_construct FROM iis) AND v.id_parent = n.id AND v.id_topicmap = 0UNION SELECT id, id_parent, 0 AS other, '' AS type FROM roles WHERE id IN (SELECT id_construct FROM iis) AND id_topicmap = 0 UNION SELECT id, 0 AS id_parent, 0 AS other, '' AS type FROM topicmaps WHERE id IN (SELECT id_construct FROM iis); Show examples
8	**0.00**	4	0.0s	SELECT id FROM roles WHERE id_parent = 0 AND id_type = 0; Show examples

9	**0.00**	1	0.0s	**SELECT DISTINCT** id_type **FROM** roles **WHERE** id_topicmap = 73060;
10	**0.00**	1	0.0s	**SELECT DISTINCT** id_type **FROM** associations **WHERE** id_topicmap = 73060;
11	**0.00**	109	0.4s	**SELECT** l.id,reference **FROM** locators **AS** l, rel_item_identifiers **AS** r **WHERE** r.id_construct = 0 **AND** r.id_locator = l.id; Show examples
12	**0.00**	1	0.0s	**SELECT** id **FROM** topics **WHERE** id_topicmap = 73060;
13	**0.00**	112	0.3s	**SELECT** t.id **FROM** topics **AS** t, locators **AS** l, rel_subject_identifiers **AS** r **WHERE** id_topicmap = 0 **AND** reference = '' **AND** l.id = r.id_locator **AND** r.id_topic = t.id; Show examples
14	**0.00**	2	0.0s	**SELECT NULL AS** TABLE_CAT, n.nspname **AS** TABLE_SCHEM, c.relname **AS** TABLE_NAME, **CASE** n.nspname ~ '' **OR** n.nspname = '' **WHEN** true **THEN CASE WHEN** n.nspname = '' **OR** n.nspname = '' **THEN CASE** c.relkind **WHEN** '' **THEN** '' **WHEN** '' **THEN** '' **WHEN** '' **THEN** '' **ELSE NULL END WHEN** n.nspname = '' **THEN CASE** c.relkind **WHEN** '' **THEN** '' **WHEN** '' **THEN** '' **ELSE NULL END ELSE CASE** c.relkind **WHEN** '' **THEN** '' **WHEN** '' **THEN** '' **ELSE NULL END END WHEN** false **THEN CASE** c.relkind **WHEN** '' **THEN** '' **WHEN** '' **THEN** '' **WHEN** '' **THEN** '' **WHEN** '' **THEN** '' **ELSE NULL END ELSE NULL END AS** TABLE_TYPE, d.description **AS** REMARKS **FROM** pg_catalog.pg_namespace n, pg_catalog.pg_class c **LEFT JOIN** pg_catalog.pg_description d **ON** (c.oid = d.objoid **AND** d.objsubid = 0) **LEFT JOIN** pg_catalog.pg_class dc **ON** (d.classoid=dc.oid **AND** dc.relname='') **LEFT JOIN** pg_catalog.pg_namespace dn **ON** (dn.oid=dc.relnamespace **AND** dn.nspname='') **WHERE** c.relnamespace = n.oid **AND** (false **OR** (c.relkind = '' **AND** n.nspname !~ '' **AND** n.nspname <> '')) **ORDER BY** TABLE_TYPE,TABLE_SCHEM,TABLE_NAME; Show examples
15	**0.00**	1	0.0s	**SELECT DISTINCT** id_type **FROM** names **WHERE** id_topicmap = 73060;
16	**0.00**	1	0.0s	**SELECT** id_theme **FROM** rel_themes **WHERE** id_scope **IN** (**SELECT** id_scope **FROM** variants **WHERE** id_topicmap = 73060) **OR** id_scope **IN** (**SELECT** id_scope **FROM** names **WHERE** id_topicmap = 73060 **AND** id **IN** (**SELECT** id_parent **FROM** variants));
17	**0.00**	1	0.0s	**SELECT** id_theme **FROM** rel_themes **WHERE** id_scope **IN** (**SELECT** id_scope **FROM** names **WHERE** id_topicmap = 73060);
18	**0.00**	1	0.0s	**SELECT** id_theme **FROM** rel_themes **WHERE** id_scope **IN** (**SELECT** id_scope **FROM** associations **WHERE** id_topicmap = 73060);
19	**0.00**	1	0.0s	**SELECT** id_theme **FROM** rel_themes **WHERE** id_scope **IN** (**SELECT** id_scope **FROM** occurrences **WHERE** id_topicmap = 73060);
20	**0.00**	1	0.0s	**SELECT** id **FROM** associations **WHERE** id_topicmap = 73060;

A.2 MaJorToM PostgreSQL Log Analysis Report (After Optimization)

pgFouine: PostgreSQL log analysis report

Overall statistics | Queries by type | Queries that took up the most time (N) | Slowest queries | Most frequent queries (N) | Slowest queries (N)

Normalized reports are marked with a "(N)".

- Generated on 2010-10-25 15:43
- Parsed C:\Program Files (x86)\PostgreSQL\8.4\data\pg_log\postgresql-2010-10-25.log (16,339 lines) in 30s
- Log from 2010-10-25 15:41:16 to 2010-10-25 15:42:47

Overall statistics ^

- Number of unique normalized queries: 52
- Number of queries: 4,058
- Total query duration: 2.8s
- First query: 2010-10-25 15:41:16
- Last query: 2010-10-25 15:42:47
- Query peak: 1,984 queries/s at 2010-10-25 15:41:34

Queries by type ^

Type	Count	Percentage
SELECT	3,887	95.8
INSERT	160	3.9

Queries that took up the most time (N) ^

Rank	Total duration	Times executed	Av. duration (s)	Query
1	**2.3s**	243	0.01	`SELECT id_type FROM typeables AS ty, topics AS t WHERE ty.id = 0;` Show examples
2	**0.2s**	103	0.00	`SELECT id FROM occurrences WHERE id_parent = 0;` Show examples
3	**0.1s**	1,987	0.00	`SELECT DISTINCT id_theme FROM rel_themes WHERE id_scope = 0;` Show examples
4	**0.1s**	116	0.00	`SELECT t.id FROM topics AS t, locators AS l, rel_subject_identifiers AS r WHERE id_topicmap = 0 AND reference = '' AND l.id = r.id_locator AND r.id_topic = t.id;` Show examples
5	**0.1s**	5	0.01	`WITH ids AS (SELECT id FROM reifiables WHERE id_reifier = 0)SELECT id, id_parent, 0 AS other, '' AS type FROM associations WHERE id IN (SELECT id FROM ids) UNION SELECT id, id_parent, 0 AS other, '' AS type FROM names WHERE id IN (SELECT id FROM ids) UNION SELECT id, id_parent, 0 AS other, '' AS type FROM occurrences WHERE id IN (SELECT id FROM ids) UNION SELECT v.id, v.id_parent, n.id_parent, '' AS type FROM variants AS v, names AS n WHERE v.id IN (SELECT id FROM ids) AND v.id_parent = n.id UNION SELECT id, id_parent, 0 AS other, '' AS type FROM roles WHERE id IN (SELECT id FROM ids) UNION SELECT id, 0 AS id_parent, 0 AS other, '' AS type FROM topicmaps WHERE id IN (SELECT id FROM ids);` Show examples
6	**0.0s**	22	0.00	`SELECT id_scope FROM names WHERE id_topicmap = 0;` Show examples
7	**0.0s**	1	0.03	`SELECT DISTINCT id_type FROM rel_instance_of, topics WHERE id_topicmap = 73060 AND id = id_type AND id_instance IN ( SELECT id FROM topics AS t WHERE t.id = id_instance );`
8	**0.0s**	52	0.00	`SELECT n.nspname,c.relname,a.attname,a.atttypid,a.attnotnull,a.atttypmod,a.attlen,a.attnum,pg_catalog.pg_get_expr(def.adbin, def.adrelid) AS adsrc,dsc.description,t.typbasetype,t.typtype FROM pg_catalog.pg_namespace n JOIN pg_catalog.pg_class c ON (c.relnamespace = n.oid) JOIN pg_catalog.pg_attribute a ON (a.attrelid=c.oid) JOIN pg_catalog.pg_type t ON (a.atttypid = t.oid) LEFT JOIN pg_catalog.pg_attrdef def ON (a.attrelid=def.adrelid AND a.attnum = def.adnum) LEFT JOIN pg_catalog.pg_description dsc ON (c.oid=dsc.objoid AND a.attnum = dsc.objsubid) LEFT JOIN pg_catalog.pg_class dc ON (dc.oid=dsc.classoid AND dc.relname='') LEFT JOIN pg_catalog.pg_namespace dn ON (dc.relnamespace=dn.oid AND dn.nspname='') WHERE a.attnum > 0 AND NOT a.attisdropped AND c.relname LIKE '' ORDER BY nspname,relname,attnum;` Show examples
9	**0.0s**	156	0.00	`INSERT INTO locators (reference) SELECT '' WHERE NOT EXISTS (SELECT id FROM locators WHERE reference = '') RETURNING *;` Show examples
10	**0.0s**	92	0.00	`SELECT DISTINCT value FROM literals WHERE id = 0;` Show examples
11	**0.0s**	156	0.00	`SELECT l.id,reference FROM locators AS l, rel_subject_identifiers AS r, topics AS t WHERE r.id_topic = 0 AND r.id_locator = l.id AND t.id = r.id_topic;` Show examples
12	**0.0s**	369	0.00	`SELECT id FROM names WHERE id_parent = 0 AND id_scope = 0;` Show examples
13	**0.0s**	1	0.01	`SELECT DISTINCT id_type FROM roles WHERE id_topicmap = 73060;`
14	**0.0s**	207	0.00	`SELECT id FROM names WHERE id_parent = 0;` Show examples

15	**0.0s**	59	0.00	`SELECT id FROM roles WHERE id_parent = 0;` Show examples
16	**0.0s**	40	0.00	`SELECT id_reifier FROM reifiables WHERE id = 0;` Show examples
17	**0.0s**	109	0.00	`SELECT l.id,reference FROM locators AS l, rel_item_identifiers AS r WHERE r.id_construct = 0 AND r.id_locator = l.id;` Show examples
18	**0.0s**	111	0.00	`SELECT l.id,reference FROM locators AS l, rel_subject_locators AS r , topics AS t WHERE r.id_topic = 0 AND r.id_locator = l.id AND t.id = r.id_topic;` Show examples
19	**0.0s**	51	0.00	`SELECT id FROM names WHERE id_parent = 0 AND id_type = 0;` Show examples
20	**0.0s**	2	0.00	`INSERT INTO topicmaps (id_base_locator) SELECT id FROM locators WHERE reference = '' AND NOT EXISTS ( SELECT tm.id FROM topicmaps AS tm, locators AS l WHERE l.reference LIKE '' AND l.id = tm.id_base_locator) RETURNING *;` Show examples

Slowest queries ^

Rank	Duration (s)	Query
1	**0.03**	`SELECT DISTINCT id_type FROM rel_instance_of, topics WHERE id_topicmap = 73060 AND id = id_type AND id_instance IN ( SELECT id FROM topics AS t WHERE t.id = id_instance );`
2	**0.02**	`SELECT id_type FROM typeables AS ty, topics AS t WHERE ty.id = 73331;`
3	**0.02**	`WITH ids AS (SELECT id FROM reifiables WHERE id_reifier = 73222)SELECT id, id_parent, 0 AS other, 'a' AS type FROM associations WHERE id IN (SELECT id FROM ids) UNION SELECT id, id_parent, 0 AS other, 'n' AS type FROM names WHERE id IN (SELECT id FROM ids) UNION SELECT id, id_parent, 0 AS other, 'o' AS type FROM occurrences WHERE id IN (SELECT id FROM ids) UNION SELECT v.id, v.id_parent, n.id_parent, 'v' AS type FROM variants AS v, names AS n WHERE v.id IN (SELECT id FROM ids) AND v.id_parent = n.id UNION SELECT id, id_parent, 0 AS other, 'r' AS type FROM roles WHERE id IN (SELECT id FROM ids) UNION SELECT id, 0 AS id_parent, 0 AS other, 'tm' AS type FROM topicmaps WHERE id IN (SELECT id FROM ids);`
4	**0.01**	`SELECT id_type FROM typeables AS ty, topics AS t WHERE ty.id = 73400;`
5	**0.01**	`SELECT id_type FROM typeables AS ty, topics AS t WHERE ty.id = 73249;`
6	**0.01**	`SELECT id_type FROM typeables AS ty, topics AS t WHERE ty.id = 73355;`
7	**0.01**	`SELECT id_type FROM typeables AS ty, topics AS t WHERE ty.id = 73151;`
8	**0.01**	`SELECT id_type FROM typeables AS ty, topics AS t WHERE ty.id = 73223;`
9	**0.01**	`SELECT id_type FROM typeables AS ty, topics AS t WHERE ty.id = 73294;`
10	**0.01**	`SELECT id_type FROM typeables AS ty, topics AS t WHERE ty.id = 73423;`
11	**0.01**	`SELECT id_type FROM typeables AS ty, topics AS t WHERE ty.id = 73067;`
12	**0.01**	`SELECT id_type FROM typeables AS ty, topics AS t WHERE ty.id = 73068;`
13	**0.01**	`SELECT id_type FROM typeables AS ty, topics AS t WHERE ty.id = 73069;`
14	**0.01**	`SELECT id_type FROM typeables AS ty, topics AS t WHERE ty.id = 73383;`
15	**0.01**	`SELECT id_type FROM typeables AS ty, topics AS t WHERE ty.id = 73361;`
16	**0.01**	`SELECT id_type FROM typeables AS ty, topics AS t WHERE ty.id = 73075;`
17	**0.01**	`SELECT id_type FROM typeables AS ty, topics AS t WHERE ty.id = 73352;`
18	**0.01**	`SELECT id_type FROM typeables AS ty, topics AS t WHERE ty.id = 73406;`
19	**0.01**	`SELECT id_type FROM typeables AS ty, topics AS t WHERE ty.id = 73356;`
20	**0.01**	`SELECT id_type FROM typeables AS ty, topics AS t WHERE ty.id = 73182;`

Most frequent queries (N) ^

Rank	Times executed	Total duration	Av. duration (s)	Query
1	**1,987**	0.1s	0.00	`SELECT DISTINCT id_theme FROM rel_themes WHERE id_scope = 0;` Show examples
2	**369**	0.0s	0.00	`SELECT id FROM names WHERE id_parent = 0 AND id_scope = 0;` Show examples
3	**243**	2.3s	0.01	`SELECT id_type FROM typeables AS ty, topics AS t WHERE ty.id = 0;` Show examples
4	**207**	0.0s	0.00	`SELECT id FROM names WHERE id_parent = 0;` Show examples
5	**156**	0.0s	0.00	`INSERT INTO locators (reference) SELECT '' WHERE NOT EXISTS (SELECT id FROM locators WHERE reference = '') RETURNING *;` Show examples
6	**156**	0.0s	0.00	`SELECT l.id,reference FROM locators AS l, rel_subject_identifiers AS r, topics AS t WHERE r.id_topic = 0 AND r.id_locator = l.id AND t.id = r.id_topic;` Show examples
7	**116**	0.1s	0.00	`SELECT t.id FROM topics AS t, locators AS l, rel_subject_identifiers AS r WHERE id_topicmap = 0 AND reference = '' AND l.id = r.id_locator AND r.id_topic = t.id;` Show examples
8	**111**	0.0s	0.00	`SELECT l.id,reference FROM locators AS l, rel_subject_locators AS r , topics AS t WHERE r.id_topic = 0 AND r.id_locator = l.id AND t.id = r.id_topic;` Show examples
9	**109**	0.0s	0.00	`SELECT l.id,reference FROM locators AS l, rel_item_identifiers AS r WHERE r.id_construct = 0 AND r.id_locator = l.id;` Show examples
10	**104**	0.0s	0.00	`SELECT id, id_parent FROM roles WHERE id_player = 0;` Show examples
11	**103**	0.2s	0.00	`SELECT id FROM occurrences WHERE id_parent = 0;` Show examples
12	**92**	0.0s	0.00	`SELECT DISTINCT value FROM literals WHERE id = 0;` Show examples
13	**59**	0.0s	0.00	`SELECT id FROM roles WHERE id_parent = 0;` Show examples

14	**52**	0.0s	0.00	**SELECT** n.nspname,c.relname,a.attname,a.atttypid,a.attnotnull,a.atttypmod,a.attlen,a.attnum,pg_catalog.pg_get_expr(def.adbin, def.adrelid) **AS** adsrc,dsc.description,t.typbasetype,t.typtype **FROM** pg_catalog.pg_namespace n **JOIN** pg_catalog.pg_class c **ON** (c.relnamespace = n.oid) **JOIN** pg_catalog.pg_attribute a **ON** (a.attrelid=c.oid) **JOIN** pg_catalog.pg_type t **ON** (a.atttypid = t.oid) **LEFT JOIN** pg_catalog.pg_attrdef def **ON** (a.attrelid=def.adrelid **AND** a.attnum = def.adnum) **LEFT JOIN** pg_catalog.pg_description dsc **ON** (c.oid=dsc.objoid **AND** a.attnum = dsc.objsubid) **LEFT JOIN** pg_catalog.pg_class dc **ON** (dc.oid=dsc.classoid **AND** dc.relname='') **LEFT JOIN** pg_catalog.pg_namespace dn **ON** (dc.relnamespace=dn.oid **AND** dn.nspname='') **WHERE** a.attnum > 0 **AND NOT** a.attisdropped **AND** c.relname **LIKE** '' **ORDER BY** nspname,relname,attnum; Show examples
15	**51**	0.0s	0.00	**SELECT** id **FROM** names **WHERE** id_parent = 0 **AND** id_type = 0; Show examples
16	**40**	0.0s	0.00	**SELECT** id_reifier **FROM** reifiables **WHERE** id = 0; Show examples
17	**22**	0.0s	0.00	**SELECT** id_scope **FROM** names **WHERE** id_topicmap = 0; Show examples
18	**10**	0.0s	0.00	**SELECT** id, id_parent **FROM** roles **WHERE** id_player = 0 **AND** id_type = 0; Show examples
19	**8**	0.0s	0.00	**SELECT** id_player **FROM** roles **WHERE** id = 0; Show examples
20	**7**	0.0s	0.00	**SELECT DISTINCT** id_scope **FROM** scopeables **WHERE** id = 0; Show examples

Slowest queries (N) ^

Rank	Av. duration (s)	Times executed	Total duration	Query
1	**0.03**	1	0.0s	**SELECT DISTINCT** id_type **FROM** rel_instance_of, topics **WHERE** id_topicmap = 73060 **AND** id = id_type **AND** id_instance **IN** (**SELECT** id **FROM** topics **AS** t **WHERE** t.id = id_instance);
2	**0.01**	5	0.1s	**WITH** ids **AS** (**SELECT** id **FROM** reifiables **WHERE** id_reifier = 0)**SELECT** id, id_parent, 0 **AS** other, '' **AS** type **FROM** associations **WHERE** id **IN** (**SELECT** id **FROM** ids) UNION **SELECT** id, id_parent, 0 **AS** other, '' **AS** type **FROM** names **WHERE** id **IN** (**SELECT** id **FROM** ids) UNION **SELECT** id, id_parent, 0 **AS** other, '' **AS** type **FROM** occurrences **WHERE** id **IN** (**SELECT** id **FROM** ids) UNION **SELECT** v.id, v.id_parent, n.id_parent, '' **AS** type **FROM** variants **AS** v, names **AS** n **WHERE** v.id **IN** (**SELECT** id **FROM** ids) **AND** v.id_parent = n.id UNION **SELECT** id, id_parent, 0 **AS** other, '' **AS** type **FROM** roles **WHERE** id **IN** (**SELECT** id **FROM** ids) UNION **SELECT** id, 0 **AS** id_parent, 0 **AS** other, '' **AS** type **FROM** topicmaps **WHERE** id **IN** (**SELECT** id **FROM** ids); Show examples
3	**0.01**	243	2.3s	**SELECT** id_type **FROM** typeables **AS** ty, topics **AS** t **WHERE** ty.id = 0; Show examples
4	**0.01**	1	0.0s	**SELECT DISTINCT** id_type **FROM** roles **WHERE** id_topicmap = 73060;
5	**0.00**	1	0.0s	**SELECT DISTINCT** id_type **FROM** associations **WHERE** id_topicmap = 73060;
6	**0.00**	1	0.0s	**SELECT** id_theme **FROM** rel_themes **WHERE** id_scope **IN** (**SELECT** id_scope **FROM** variants **WHERE** id_topicmap = 73060) **OR** id_scope **IN** (**SELECT** id_scope **FROM** names **WHERE** id_topicmap = 73060 **AND** id **IN** (**SELECT** id_parent **FROM** variants));
7	**0.00**	1	0.0s	**SELECT** id **FROM** associations **WHERE** id_topicmap = 73060 **AND** id_scope **IN** (**SELECT** id_scope **FROM** rel_themes **WHERE** id_theme = 73222);
8	**0.00**	1	0.0s	**SELECT** id_theme **FROM** rel_themes **WHERE** id_scope **IN** (**SELECT** id_scope **FROM** associations **WHERE** id_topicmap = 73060);
9	**0.00**	1	0.0s	**SELECT** id, id_parent **FROM** names **WHERE** id_topicmap = 73060 **AND** id_type = 73222;
10	**0.00**	1	0.0s	**SELECT** id **FROM** associations **WHERE** id_topicmap = 73060;
11	**0.00**	1	0.0s	**SELECT** id_theme **FROM** rel_themes **WHERE** id_scope **IN** (**SELECT** id_scope **FROM** names **WHERE** id_topicmap = 73060);
12	**0.00**	1	0.0s	**SELECT** id, id_parent **FROM** occurrences **WHERE** id_topicmap = 73060 **AND** id_scope **IN** (**SELECT** id_scope **FROM** rel_themes **WHERE** id_theme = 73222);
13	**0.00**	1	0.0s	**SELECT** id, id_parent **FROM** names **WHERE** id_topicmap = 73060 **AND** id_scope **IN** (**SELECT** id_scope **FROM** rel_themes **WHERE** id_theme = 73222);
14	**0.00**	1	0.0s	**SELECT DISTINCT** id_type **FROM** occurrences **WHERE** id_topicmap = 73060;
15	**0.00**	1	0.0s	**SELECT DISTINCT** id_type **FROM** names **WHERE** id_topicmap = 73060;
16	**0.00**	1	0.0s	**SELECT** id **FROM** associations **WHERE** id_topicmap = 73060 **AND** id_type = 73222;
17	**0.00**	1	0.0s	**SELECT** id_theme **FROM** rel_themes **WHERE** id_scope **IN** (**SELECT** id_scope **FROM** occurrences **WHERE** id_topicmap = 73060);
18	**0.00**	103	0.2s	**SELECT** id **FROM** occurrences **WHERE** id_parent = 0; Show examples
19	**0.00**	22	0.0s	**SELECT** id_scope **FROM** names **WHERE** id_topicmap = 0; Show examples
20	**0.00**	2	0.0s	**INSERT INTO** topicmaps (id_base_locator) **SELECT** id **FROM** locators **WHERE** reference = '' **AND NOT EXISTS** (**SELECT** tm.id **FROM** topicmaps **AS** tm, locators **AS** l **WHERE** l.reference **LIKE** '' **AND** l.id = tm.id_base_locator) RETURNING *; Show examples

A.3 MaJorToM PostgreSQL Mapping for Simple TMAPI Calls

Duration	PostgreSQL statement
3 ms	INSERT INTO locators (reference) SELECT 'http://topic-map-under-test/bar' WHERE NOT EXISTS (SELECT id FROM locators WHERE reference = 'http://topic-map-under-test/bar') RETURNING *
3 ms	INSERT INTO topics(id_topicmap, id_parent) VALUES (13434,13434) RETURNING *
1 ms	INSERT INTO locators (reference) SELECT 'http://topic-map-under-test/bar' WHERE NOT EXISTS (SELECT id FROM locators WHERE reference = 'http://topic-map-under-test/bar')
2 ms	INSERT INTO rel_item_identifiers(id_construct, id_locator) SELECT 13437 , id FROM locators WHERE reference LIKE 'http://topic-map-under-test/bar'
4 ms	INSERT INTO rel_instance_of(id_instance, id_type) SELECT 13437,13436 WHERE NOT EXISTS (SELECT id_instance, id_type FROM rel_instance_of WHERE id_instance = 13437 AND id_type = 13436)
4 ms	INSERT INTO locators (reference) SELECT 'http://psi.topicmaps.org/iso13250/model/type-instance' WHERE NOT EXISTS (SELECT id FROM locators WHERE reference = 'http://psi.topicmaps.org/iso13250/model/type-instance') RETURNING *
4 ms	SELECT t.id FROM topics AS t, locators AS l, rel_subject_identifiers as r WHERE id_topicmap = 13434 AND reference = 'http://psi.topicmaps.org/iso13250/model/type-instance' AND l.id = r.id_locator AND r.id_topic = t.id
3 ms	INSERT INTO topics(id_topicmap, id_parent) VALUES (13434,13434) RETURNING *
2 ms	INSERT INTO locators (reference) SELECT 'http://psi.topicmaps.org/iso13250/model/type-instance' WHERE NOT EXISTS (SELECT id FROM locators WHERE reference = 'http://psi.topicmaps.org/iso13250/model/type-instance')
2 ms	INSERT INTO rel_subject_identifiers(id_topic, id_locator) SELECT 13438 , id FROM locators WHERE reference LIKE 'http://psi.topicmaps.org/iso13250/model/type-instance'
4 ms	INSERT INTO associations(id_topicmap, id_parent, id_type) VALUES (13438,13438,13438) RETURNING *
6 ms	INSERT INTO locators (reference) SELECT 'http://psi.topicmaps.org/iso13250/model/type-instance' WHERE NOT EXISTS (SELECT id FROM locators WHERE reference = 'http://psi.topicmaps.org/iso13250/model/type-instance') RETURNING *
5 ms	SELECT t.id FROM topics AS t, locators AS l, rel_subject_identifiers as r WHERE id_topicmap = 13434 AND reference = 'http://psi.topicmaps.org/iso13250/model/instance' AND l.id = r.id_locator AND r.id_topic = t.id
4 ms	INSERT INTO topics(id_topicmap, id_parent) VALUES (13434,13434) RETURNING *
2 ms	INSERT INTO locators (reference) SELECT 'http://psi.topicmaps.org/iso13250/model/instance' WHERE NOT EXISTS (SELECT id FROM locators WHERE reference = 'http://psi.topicmaps.org/iso13250/model/instance')
1 ms	INSERT INTO rel_subject_identifiers(id_topic, id_locator) SELECT 13440 , id FROM locators WHERE reference LIKE 'http://psi.topicmaps.org/iso13250/model/instance'
1 ms	INSERT INTO roles(id_topicmap, id_parent, id_type, id_player) VALUES (13434,13439,13440,13437) RETURNING *
3 ms	INSERT INTO locators (reference) SELECT 'http://psi.topicmaps.org/iso13250/model/type' WHERE NOT EXISTS (SELECT id FROM locators WHERE reference = 'http://psi.topicmaps.org/iso13250/model/type') RETURNING *
5 ms	SELECT t.id FROM topics AS t, locators AS l, rel_subject_identifiers as r WHERE id_topicmap = 13434 AND reference = 'http://psi.topicmaps.org/iso13250/model/type' AND l.id = r.id_locator AND r.id_topic = t.id
4 ms	INSERT INTO topics(id_topicmap, id_parent) VALUES (13434,13434) RETURNING *
1 ms	INSERT INTO locators (reference) SELECT 'http://psi.topicmaps.org/iso13250/model/type' WHERE NOT EXISTS (SELECT id FROM locators WHERE reference = 'http://psi.topicmaps.org/iso13250/model/type')
3 ms	INSERT INTO rel_subject_identifiers(id_topic, id_locator) SELECT 13442 , id FROM locators WHERE reference LIKE 'http://psi.topicmaps.org/iso13250/model/type'
1 ms	INSERT INTO roles(id_topicmap, id_parent, id_type, id_player) VALUES (13434,13439,13442,13436) RETURNING *

Table A.1: Count and duration of SQL statements for invocation of `TopicMap.createTopic(type)` against TMAPI-based database back-end

Duration	PostgreSQL statement
3 ms	INSERT INTO locators (reference) SELECT 'http://topic-map-under-test/foo' WHERE NOT EXISTS (SELECT id FROM locators WHERE reference = 'http://topic-map-under-test/foo') RETURNING *
3 ms	INSERT INTO topics(id_topicmap, id_parent) VALUES (13434,13434) RETURNING *
2 ms	INSERT INTO locators (reference) SELECT 'http://topic-map-under-test/foo' WHERE NOT EXISTS (SELECT id FROM locators WHERE reference = 'http://topic-map-under-test/foo')
2 ms	INSERT INTO rel_item_identifiers(id_construct, id_locator) SELECT 13436 , id FROM locators WHERE reference LIKE 'http://topic-map-under-test/foo'
3 ms	SELECT l.id,reference FROM locators AS l, rel_subject_identifiers AS r, topics AS t WHERE r.id_topic = 13436 AND r.id_locator = l.id AND t.id = r.id_topic
4 ms	SELECT l.id,reference FROM locators AS l, rel_subject_locators AS r , topics AS t WHERE r.id_topic = 13436 AND r.id_locator = l.id AND t.id = r.id_topic
3 ms	SELECT l.id,reference FROM locators AS l, rel_item_identifiers AS r WHERE r.id_construct = 13436 AND r.id_locator = l.id

Table A.2: Count and duration of SQL statements for invocation of `TopicMap.createTopic()` against TMAPI-based database back-end

A.4 JTMQR 2.0 Schema

```
{
  "$schema" : "http://json-schema.org/draft-03/schema#",
  "id" : "http://code.google.com/p/tmql/wiki/JTMQR2_0#",

  "type":"object",
  "title":"A JTMQR 2.0 document",
  "properties":{
    "version" :{"type":"string", "required":true, "pattern":"2.0"},
    "metadata":{"type":"object", "required":true, "properties":{
      "columns":{"type":"integer", "required":true, "minimum":0},
      "rows"   :{"type":"integer", "required":true, "minimum":0},
      "headers":{"type":"array",   "required":true,
        "items":{"type":["string","null"]}
      }
    }},
    "tuples"  :{"type":"array", "required":true,
      "items":{"type":"array",
        "items":{[
          "type":[
            "string",
            "number",
            "boolean",
            "null",
            {"type":"object", "properties":{
              "jtm":{"required":true, "type":[
                "object"
                // As for JTM 1.x no schema definitions are available
                // the type of "jtm" property was set to "object" to enable
                // validation passing. Otherwise the following references
                // to JTM schemas MAY work.
                // {"$ref" : "http://www.cerny-online.com/jtm/1.0/#"},
                // {"$ref" : "http://www.cerny-online.com/jtm/1.1/#"}
              ]}
            }}
          ]
        ]}
      }
    },
    "ordered"   :{"type":"boolean", "optional":true}
  }
}
```

Listing A.1: JSON Schema of JTMQR 2.0.

A.5 Application Context Configuration for the Topic Maps Query Service

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

  <!-- Topic Map System -->
  <bean id="topicMapSystemFactory"
        class="org.tmapi.core.TopicMapSystemFactory"
        factory-method="newInstance"/>
  <bean id="topicMapSystem" class="org.tmapi.core.TopicMapSystem"
        factory-bean="topicMapSystemFactory"
```

```
13          factory-method="newTopicMapSystem"/>
14
15    <!-- Topic Map Query Service -->
16    <bean id="topicMapQueryService"
17          class="de.topicmapslab.remoting.tmqs.TopicMapQueryServiceImpl">
18      <constructor-arg name="topicMapSystem"
19                       ref="topicMapSystem" />
20    </bean>
21
22    <!-- RabbitMQ configuration -->
23    <bean id="rabbitmqConnectionFactory"
24          class="com.rabbitmq.client.ConnectionFactory">
25        <!--<property name="Host" value="localhost" />-->
26        <!--<property name="Username" value="guest" />-->
27        <!--<property name="Password" value="guest" />-->
28    </bean>
29
30    <bean id="rabbitmqConnection"
31          class="com.rabbitmq.client.Connection"
32          factory-bean="rabbitmqConnectionFactory"
33          factory-method="newConnection"
34          destroy-method="close" />
35
36    <bean id="tmqsChannel"
37          class="com.rabbitmq.client.AMQP.Channel"
38          factory-bean="rabbitmqConnection"
39          factory-method="createChannel"/>
40
41    <!-- TMQS Consumer -->
42    <bean id="tmqsConsumer"
43          class="de.topicmapslab.remoting.rabbitmq.ReplyingJsonConsumer">
44      <constructor-arg name="channel" ref="tmqsChannel" />
45      <constructor-arg name="handler">
46        <bean id="queryActionHandler"
47              class="de.topicmapslab.remoting.adapters.QueryActionHandler">
48          <constructor-arg ref="topicMapQueryService" />
49        </bean>
50      </constructor-arg>
51      <constructor-arg name="queueName"    value="tmlab:tmqs:query" />
52      <constructor-arg name="exchangeName" value="tmlab:tmqs:query" />
53      <constructor-arg name="exchangeType" value="fanout" />
54      <constructor-arg name="routingKey"   value="" />
55    </bean>
56  </beans>
```

Listing A.2: Spring configuration for TMQS application context.

A.6 TMQL Expression Tree Sample

Using the TMQL Console[1] or the TMQL Canonizer[2] one can produce the textual representation of an TMQL4J expression tree for given TMQL statements. For instance, the expression tree[3] of the following sample query:

```
1  # Find all pokemons weighting more than 200 lbs
2  %prefix p http://testmap/pokemon/%23
3
```

[1] see https://github.com/mhoyer/tmql-console

[2] see http://canonizer.topicmapslab.de/

[3] The output is based on TMQL4J version 3.1.0 and required some cleaning. Hence it does not *entirely* match the original output.

```
p:pokemon >> instances
  [ . >> characteristics p:weight > 200 ]
```

Listing A.3: Sample query to demonstrate TMQL expression tree generation

```
QueryExpression(%prefix p http://testmap/pokemon/%23
 |              p:Pokemon >> instances
 |              [ . >> characteristics p:weight > 200 ])
 + EnvironmentClause(%prefix p http://testmap/pokemon/%23)
 | + PrefixDirective(%prefix p http://testmap/pokemon/%23)
 + PathExpression(p:Pokemon >> instances
  |               [ . >> characteristics p:weight > 200 ])
  + PostfixedExpression(p:Pokemon >> instances
   |                    [ . >> characteristics p:weight > 200 ])
   + SimpleContent(p:Pokemon >> instances
    |              [ . >> characteristics p:weight > 200 ])
    + Anchor(p:Pokemon)
    + Navigation(>> instances [ . >> characteristics p:weight > 200 ])
     + StepDefinition(>> instances [ . >> characteristics p:weight > 200 ])
      + Step(>> instances)
      + FilterPostfix([ . >> characteristics p:weight > 200 ])
       + BooleanExpression(. >> characteristics p:weight > 200)
        + BooleanPrimitive(. >> characteristics p:weight > 200)
         + ExistsClause(. >> characteristics p:weight > 200)
          + Content(. >> characteristics p:weight > 200)
           + QueryExpression(. >> characteristics p:weight > 200)
            + PathExpression(. >> characteristics p:weight > 200)
             + PostfixedExpression(. >> characteristics p:weight > 200)
              + TupleExpression(. >> characteristics p:weight > 200)
               + AliasValueExpression(. >> characteristics p:weight > 200)
                + ValueExpression(. >> characteristics p:weight > 200)
                 + ValueExpression(. >> characteristics p:weight)
                 | + Content(. >> characteristics p:weight)
                 |  + QueryExpression(. >> characteristics p:weight)
                 |   + PathExpression(. >> characteristics p:weight)
                 |    + PostfixedExpression(. >> characteristics p:weight)
                 |     + SimpleContent(. >> characteristics p:weight)
                 |      + Anchor(.)
                 |      + Navigation(>> characteristics p:weight)
                 |       + StepDefinition(>> characteristics p:weight)
                 |        + Step(>> characteristics p:weight)
                 |         + Anchor(p:weight)
                 + ValueExpression(200)
                  + Content(200)
                   + QueryExpression(200)
                    + PathExpression(200)
                     + PostfixedExpression(200)
                      + SimpleContent(200)
                       + Anchor(200)
```

Listing A.4: Textual representation of an expression tree.

Declaration

I declare that the submitted work has been completed by me the undersigned and that I have not used any other than permitted reference sources or materials nor engaged in any plagiarism. All references and other sources used by me have been appropriately acknowledged in the work. I further declare that the work has not been submitted for the purpose of academic examination, either in its original or similar form, anywhere else.

Leipzig, 24th October 2011 Signature